5

PROCESS AND PRODUCT APPLICATIONS

5.1 INTRODUCTION

Process and product modeling applications cover broad and overlapping areas. The importance of good processes was introduced in Chapter 1 as a way to efficiently produce software that is useful to people. The ultimate goal is to provide product utility to users. Applications in this chapter elaborate more on process technology, covering both process life cycle and process improvement aspects along with product modeling. Substantial modeling and simulation work has been done for process applications due to the concentrated focus on process improvement in recent years.

Although software is the product that people actually use to obtain benefits from, there have been fewer software product applications with system dynamics than process applications. Some product attributes are not as amenable to tracking over time in an aggregate fashion. Discrete modeling approaches are more frequently used for product-related attributes than continuous systems modeling, as they allow linking of different attribute measures to individual software entities.

Process and product analysis are often difficult to extricate in many modeling applications, since software artifacts are primary state variables for both. Both types of applications frequently center on software artifacts and their attributes. Processes were defined generally as the life-cycle steps taken to develop software, so modeling life cycles entails the tracking of product artifacts over time. Hence, levels are dedicated to software artifacts, and a view of product characteristics or measurements is intertwined with the process view.

Software Process Dynamics. By Raymond J. Madachy

Two types of processes can be modeled to investigate product phenomena. Development process models mainly address software defect introduction and removal rates. Operational process models address the probability of various classes of product failure during usage (e.g., race conditions or missing real-time deadlines). System dynamics does not lend itself easily to failure mode assessment, so other methods like analytical approaches are frequently used instead.

Process and product factors account for a very significant amount of effort variance between projects. Product size itself has the biggest overall influence on effort (next in influence are the collective people factors; see Chapter 4). The process and product related factors (besides size) in COCOMO II with direct impacts on productivity are shown in Figure 5.1.

Product complexity constitutes several attributes and by itself provides for the largest impact of any single factor in the COCOMO model. Another important product-related factor is *Precedentedness* (with a range of 1.33), but it also has an organizational experience component to it and thus is not shown as a direct product factor. There may be additional business or mission-critical product factors, such as security or safety, that are important and large sources of variance that can also be added to COCOMO.

The life-cycle process for a project is not a source of variance in the COCOMO model, however, because it assumes that processes are well suited and properly tai-

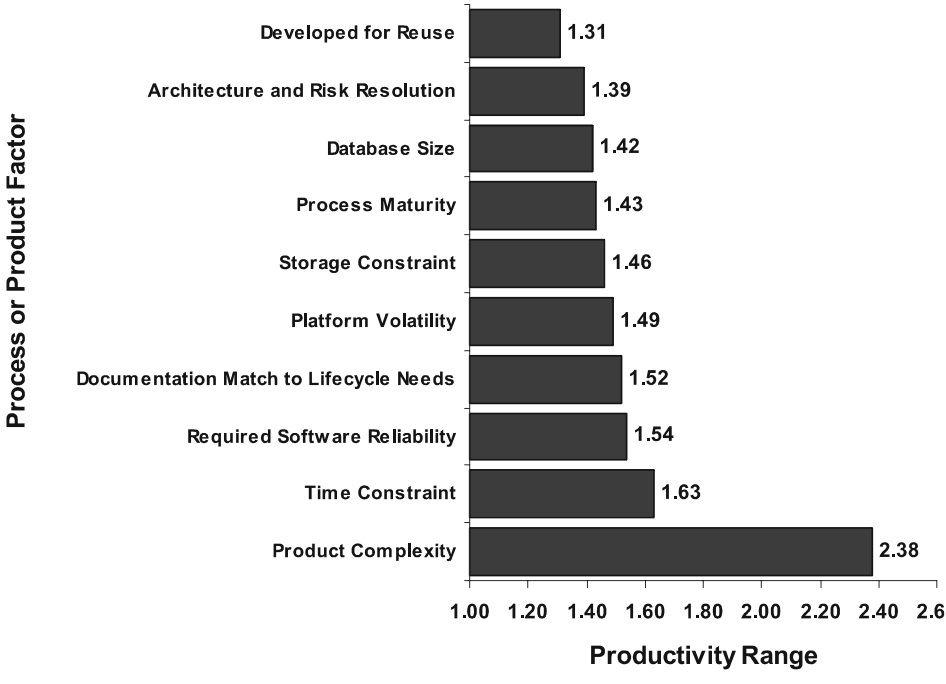


Figure 5.1. Process and product factor impacts from COCOMO II. What causes these?

lored for a project. Since this is not always the case, modeling applications like many in this chapter can be used to assess the impact of life cycles.

A combined process and product opportunity tree is shown in Figure 5.2. The tree demonstrates the many overlaps of concerns shown in the middle branches. There are also some common leaf nodes visible upon further inspection. These opportunities for

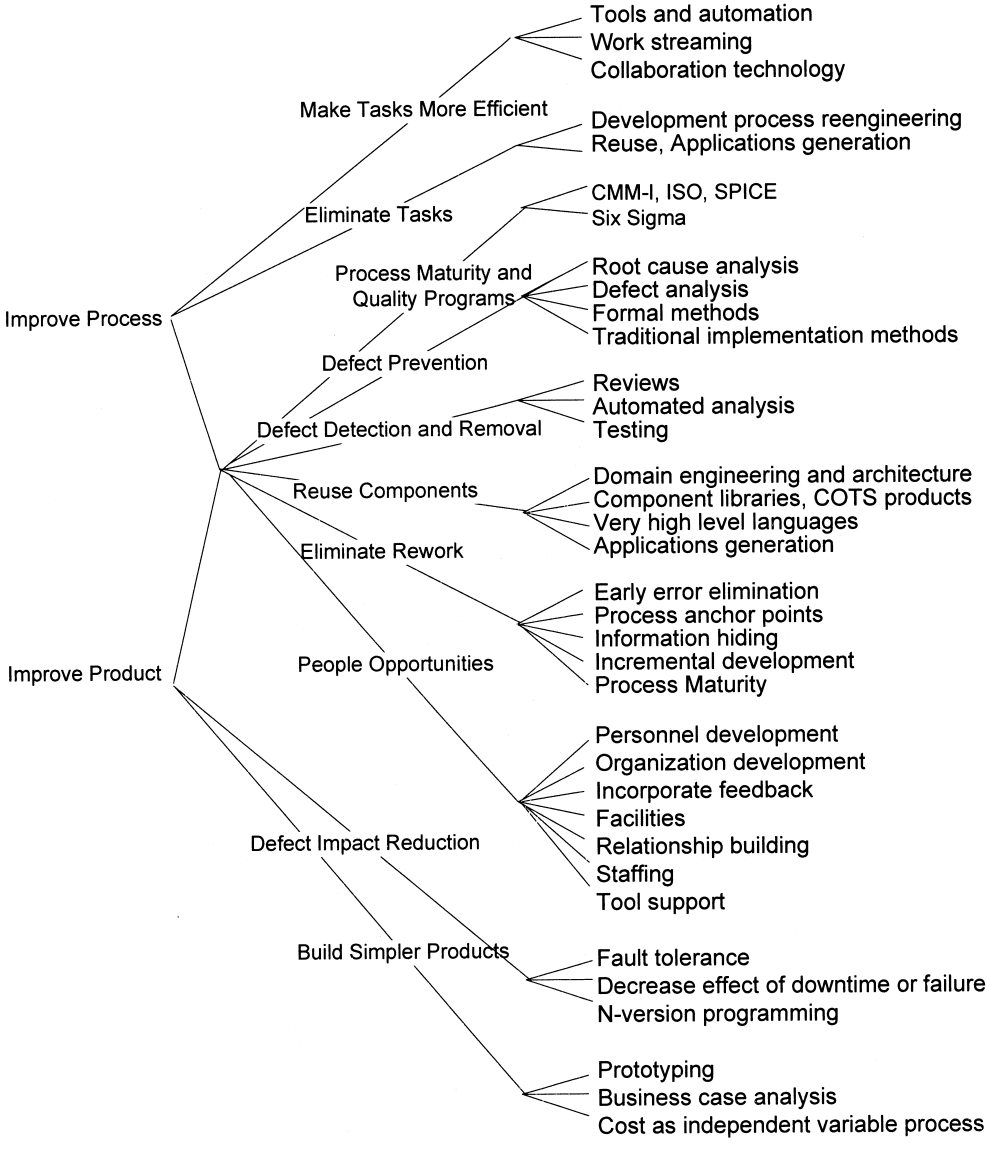


Figure 5.2. Process and product opportunity tree.

improving processes and products also represent opportunities to use modeling and simulation to better understand and leverage the strategies.

Product characteristics are wide-ranging in scope. Desired product attributes, from a user view, usually fall under the general umbrella of "quality." Various aspects of software quality are frequently considered, but the definition of quality depends on who is judging it. Defects are usually modeled to assess quality since they can impact a system with respect to quality attributes.

These quality aspects are frequently termed "ilities" and refer to nonfunctional requirements (usability, dependability, security, etc.). Some representative ilities, or product quality attributes, are shown in Figure 5.3. It is not an exhaustive list. Since quality is in the eyes of the beholder, its definition depends on one's perspective and expectations of utility from the software. The chapter section on quality provides details of using a stakeholder value-based approach for defining quality measurements.

Important relationships between process strategies and products can be exploited, because the impact of different processes and technologies on product attributes can be evaluated through modeling and simulation. A framework for the contribution of modeling to meet product goals is evaluating the effectiveness of the strategy opportunities against specific goals like aspects of quality. The opportunity tree in Figure 5.2 shows that sample strategies and product goals could be related to the attributes in Figure 5.3. Much of the difficulty lies in modeling connections between the strategies, intermediate quantities, and desired product attributes.

For example, three generic process strategies for producing quality software products are to avoid problems (defect prevention), eliminate problems (finding and fixing defects), or to reduce the impact of problems. Modeling the impact of specific types of defects on particular product attributes will help one understand how the defect strategies contribute to product goals.

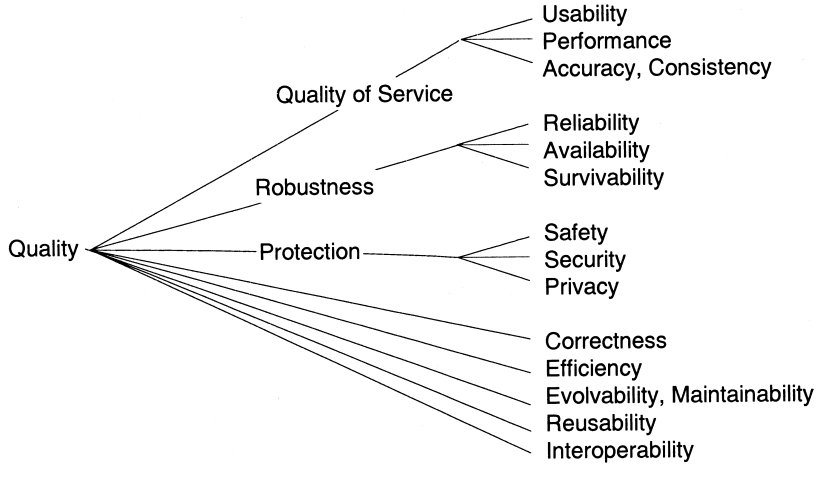


Figure 5.3. Representative quality attributes.

The existence of strong dependencies between product and process was an underlying concept in a simulation model in [Pfahl, Birk 2000]. The purpose of process–product dependency (PPD) models is to focus process improvement actions to be most effective with regard to product quality goals. The work explored how system dynamics models can be used to check the plausibility of achieving product goals when implementing improvements derived from PPD models.

Discrete-event modeling has some advantages for product analysis because different attributes can be attached to individual entities like defects. The attributes may change when system events occur during the simulation (such as a defect-detection or fixing event). Defects can be characterized by their type, severity, detection effort, removal effort, and so on. A hybrid approach combining continuous and discrete-event modeling is very attractive for product applications. It can model the creation of artifacts with attributes, modify those attributes based on system variables, and allow system variables to vary continuously.

A recent book [Acuña, Juristo 2005] contains a number of process and product modeling applications by leading researchers and practitioners. Though it does not focus on system dynamics modeling exclusively, it would serve as a good supplement to this chapter.

5.2 OVERVIEW OF APPLICATIONS

Some primary strategies identified in the opportunity tree are covered in the application sections. The chapter will start with an application for peer reviews, one of most well-known and studied processes for efficiently finding defects and improving product quality. The example from my dissertation on formal inspections [Madachy 1994b] shows substantial detail of the model and its assessment. The last subsection shows examples of using peer review metrics for model calibration.

Global process feedback takes a unique and broader view of the process by accounting for external feedback from the field. Several applications of global process modeling are identified, showing how long-term feedback models of software changes can provide good fidelity. Understanding global feedback can also help address trends such as globalization, COTS, or open-source development. A detailed example model for progressive and anti-regressive work is described in [Ramil et al. 2005].

The related methods of reuse and COTS for improving software productivity are covered next. A study by [Lo 1999] investigates the dynamics of reuse and fourth-generation languages. In the COTS application section, a detailed model of COTS glue-code development and integration is described [Kim, Baik 1999]. Additionally, an example model concept for COTS lifespan dynamics is reviewed.

Software architecting is one of the most success-critical aspects of developing software. An example comprehensive model of iterative architecting activities from [Fakharzadeh, Mehta 1999] is described, that is calibrated to extensive effort data and addresses best practices for architecting.

The next section is dedicated to defects and quality. It covers some issues of defining and modeling quality, and includes applications for defect analysis. The example

with defect removal techniques and orthogonal defect classification illustrates how the effectiveness of different methods depends on the type of defect.

Requirements volatility is a seemingly persistent phenomenon on projects. The dynamics behind product volatility are described and investigated with an example modeling application based on [Ferreira et al. 2003]. This doctoral dissertation research included the use of a comprehensive survey to obtain data for the model.

Continuous process improvement is undertaken by organizations to better meet cost, schedule, or functionality goals. Accordingly, extensive coverage of software process improvement is included, based on a comprehensive model. A process improvement model was developed by Burke [Burke 1996] and adapted for use at Xerox [Ho 1999] and other places. The Xerox application demonstrates how a model can be adapted for an organization and modified to be more user friendly. It also provides illustrative detail of defining different test cases and running sensitivity tests.

5.3 PEER REVIEWS

A major goal of software engineering is to produce higher-quality software while keeping effort expenditure and schedule time to a minimum. Peer reviews are a commonly accepted practice to achieve this. Intermediate work products are reviewed to eliminate defects when they are less costly to fix. Peer reviews of software artifacts range from informal reviews to structured walk-throughs to formal inspections. Many variations exist, involving a variety of participants.

Inspections were first devised by Fagan at IBM as a formal, efficient, and economical method of finding errors in design and code [Fagan 1976]. Inspections are carried out in a prescribed series of steps including preparation, having an inspection meeting at which specific roles are executed, and rework of errors discovered by the inspection. Standard roles include moderator, scribe, inspectors, and author.

Since their introduction, the use of inspections has been extended to requirements, testing plans, and more. Thus, an inspection-based process would perform inspections on artifacts throughout system development such as requirements descriptions, design, code, test plans, user guides, or virtually any engineering document.

Inspections are just one method of performing peer reviews, and are more rigorous than other types such as structured walk-throughs. Peer reviews can be included to some degree in virtually all other software life-cycle processes, so peer reviews can be considered a variation within the major life-cycle models.

In general, results have shown that extra effort is required during design and coding for peer reviews and much more is saved during testing and integration, resulting in a reduced overall schedule. By detecting and correcting errors in earlier stages of the process such as design, significant reductions can be made since the rework is much less costly compared to later in the testing and integration phases. Countless authors have corroborated this result over the years, and many references can be found in [Madachy 1994b].

Peer reviews in general are a good idea, but they should not be overdone. There are diminishing returns from peer reviews (see insights from the example inspection mod-

el in the next section), and they are not appropriate for finding all classes of software errors.

A number of extensive modeling efforts have focused on peer reviews as a means of finding and fixing defects, including process trade-offs. Several researchers have used system dynamics to investigate the cost, schedule, and quality impact of using formal inspections and other peer reviews on work products [Madachy 1994b, Tvedt 1995], and others have used discrete event modeling [Raffo 1995, Eickelmann et al. 2002].

In [Madachy 1994b], a process model was used to examine the effects of inspection practices on cost, schedule, and quality (defect levels) throughout the life cycle. It used system dynamics to model the interrelated flows of tasks, errors, and personnel throughout different life-cycle phases and was extensively calibrated to industrial data. This application is highlighted in the next section.

A discrete event model for analyzing the effect of inspections was developed in [Raffo 1995]. The quantitative cost/quality trade-offs for performing inspections were very close to those derived from [Madachy 1994b] at the top level. Both studies support the contention that even though inspections may increase effort early on, the overall development costs and schedule are reduced.

The [Tvedt 1995] model allows one to evaluate the impact of process improvements on cycle time. It specifically addresses concurrent incremental software development to assess the impact of software inspections. The model enables controlled experiments to answer questions such as "What kind of cycle time reduction can I expect to see if I implement inspections?" or "How much time should I spend on inspections?" It modeled the specific activities within the inspection process so that one can experiment with effort dedicated to preparation, inspection, rework, and so on.

Recent research on modeling inspections is in [Twaites et al. 2006]. This work uses a system dynamics model to assess the effect of prioritizing the items under inspection when presented with an accelerated schedule. The model is provided with this book and is listed in Appendix C.

5.3.1 Example: Modeling an Inspection-Based Process

In my dissertation, I investigated the dynamic effects of inspections with a simulation model [Madachy 1994b, Madachy 1996b]. It examines the effects of inspection practices on cost, schedule, and quality throughout the life cycle. Quality in this context refers to the absence of defects, where a defect is defined as a flaw in the specification, design, or implementation of a software product. Table 5.3-1 is a high-level summary of the model.

The model demonstrates the effects of performing inspections or not, the effectiveness of varied inspection policies, and the effects of other managerial policies such as manpower allocation. The dynamic project effects and their cumulative measures are investigated by individual phase.

Development of the inspection-based process model drew upon extensive literature search, analysis of industrial data, and expert interviews. The basic reference behavior identified is that extra effort is required during design and coding for inspections and

Table 5.1. Peer review model overview

Purpose: Process Improvement, Planning
Scope: Development Project

Inputs and Parameters	Levels	Major Relationships	Outputs
• Design inspection practice • Code inspection practice • Job size • Inspection efficiency • Design error density • Code error density • Effort constant • Schedule constant • Rework effort per design error • Rework effort per code error	• Tasks Required Designed For coding Coded Ready for test • Errors Design Undetected design Detected design Reworked design Code Detected code Undetected code Reworked code Escaped Fixed in test	• Staffing rates • Defect rates • Schedule tradeoff	• Staffing per activity • Effort per activity • Schedule per activity • Errors per type

much more is saved during testing and integration, resulting in reduced overall schedule effect. This is shown in [Fagan 1976] and represents a goal of the model; to be able to demonstrate this effect on effort and schedule as a function of inspection practices. The following sections are drawn from [Madachy 1996b], with full detail and complete references in [Madachy 1994b].

5.3.1.1 Industrial Data Collection and Analysis

Data collection and analysis for a large project using an inspection-based process was performed at Litton Data Systems. As an application for command and control, the software comprised about 500 thousand lines of code (KSLOC). Over 300 inspection data points were collected and analyzed from the project. Analysis of this and additional projects is documented in [Madachy 1995c]. Some results with important ramifications for a dynamic model include defect density and defect removal effectiveness life-cycle trends and activity distribution of effort. Calibrations are made to match the model to these defect density trends and activity distributions in selected test cases. Additional inspection data was available from JPL to support model analysis and calibration [Kelly, Sherif 1990].

Data for the previous project at Litton in the same product line and having identical environmental factors except for the use of inspections was also collected. It provides several parameters used in calibration of the system dynamics model such as defect density and the average cost to fix a defect found during testing. It also enables a comparison to judge the relative project effects of performing inspections.

Several senior personnel at Litton and international experts were interviewed to obtain data and heuristics for process delays, manpower allocation policies and general project trends relevant to inspection practices.

The model is validated against Litton data such as the proportions of effort for preparation, inspection meeting, rework and other development activities; schedule; and defect profiles. See [Madachy 1995c] for additional analysis of inspection data.

5.3.1.2 Model Overview

The model covers design through testing activities, including inspections of design and code. Figure 5.4 shows the task and error portions of the model. It is assumed that inspections and system testing are the only defect removal activities, as practiced in some organizations. The model also assumes a team trained in inspections, such that inspections can be inserted into the process without associated training costs.

Project behavior is initiated with the influx of requirements. Rates for the different phase activities are constrained by the manpower allocations and current productivity, and are integrated to determine the state variables, or levels of tasks in design, code, inspection, testing, and so on. The tasks are either inspected or not during design or code activities as a function of inspection practices.

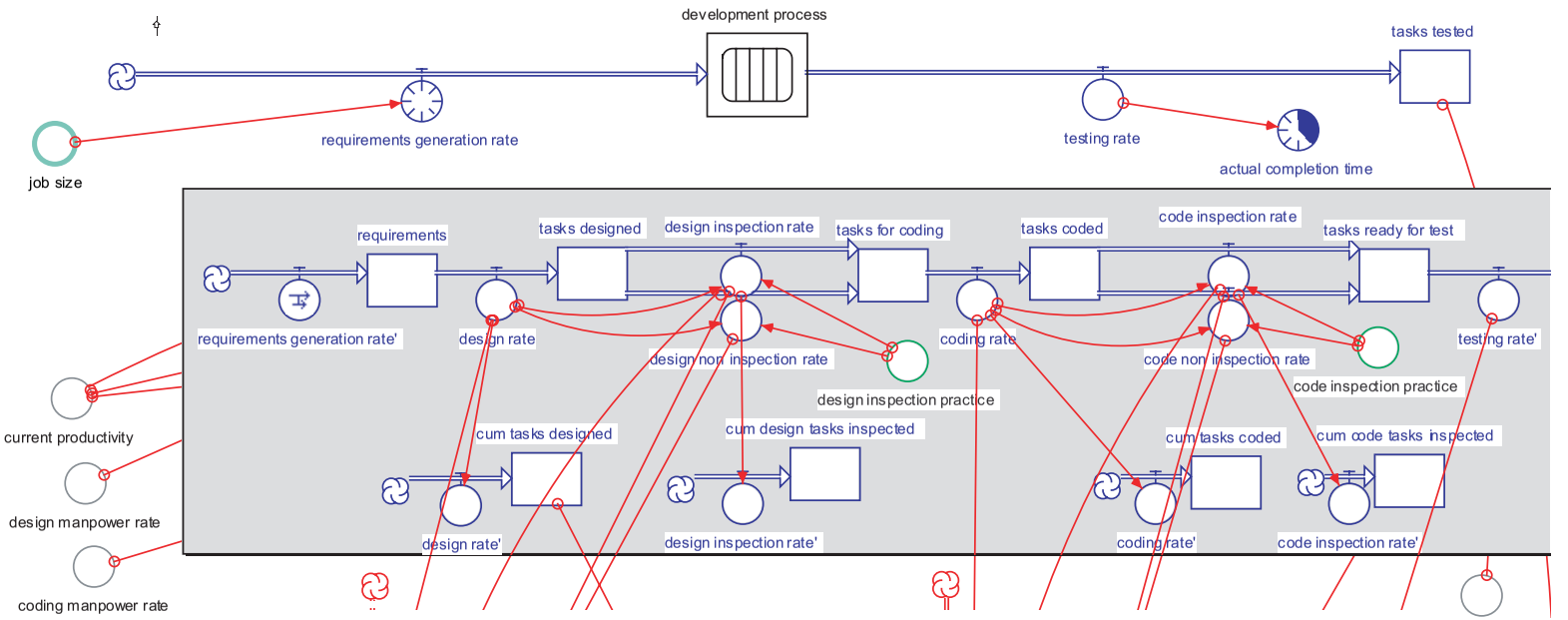
Errors are generated as a coflow of design and coding activities, and eventually detected by inspections or passed onto the next phase. The detection of errors is a function of inspection practices and inspection efficiency. All errors detected during inspections are then reworked. Note that escaped design errors are amplified into coding errors. Those errors that still escape code inspections are then detected and corrected during integration and test activities. The effort and schedule for the testing phase is adjusted for the remaining error levels.

Other sectors of the model implement management policies such as allocating personnel resources and defining staffing curves for the different activities. Input to the policy decisions are the job size, productivity, schedule constraints, and resource leveling constraints. Effort is instrumented in a cumulative fashion for all of the defined activities. Learning takes place as a function of the job completion and adjusts the productivity. Schedule compression effects are approximated similar to the COCOMO cost driver effects for relative schedule.

The actual completion time is implemented as a cycle time utility whose final value represents the actual schedule. It instruments the maximum in-process time of the tasks that are time-stamped at requirements generation rate and complete testing.

5.3.1.2.1 Model Calibration. To gauge the impact of inspections, effort expenditure was nominally calibrated to the staffing profiles of COCOMO, a widely accepted empirically based model. Correspondence between the dynamic model and COCOMO is illustrated in Figure 5.5. It is based on the phase-level effort and schedule of a 32 KSLOC embedded-mode reference project.

In the nominal case, no inspections are performed, and the manpower allocation policy employs the staffing curves in Figure 5.5 to match the rectangular COCOMO effort profiles as a calibration between the models. Thus, the area under the design


development process
tasks tested
requirements generation rate
testing rate
actual completion time
job size
requirements
tasks designed
design inspection rate
tasks for coding
tasks coded
code inspection rate
tasks ready for test
requirements generation rate'
design rate
design non inspection rate
coding rate
code non inspection rate
testing rate'
design inspection practice
code inspection practice
current productivity
cum tasks designed
cum design tasks inspected
cum tasks coded
cum code tasks inspected
design manpower rate
design rate'
design inspection rate'
coding rate'
code inspection rate'
coding manpower rate

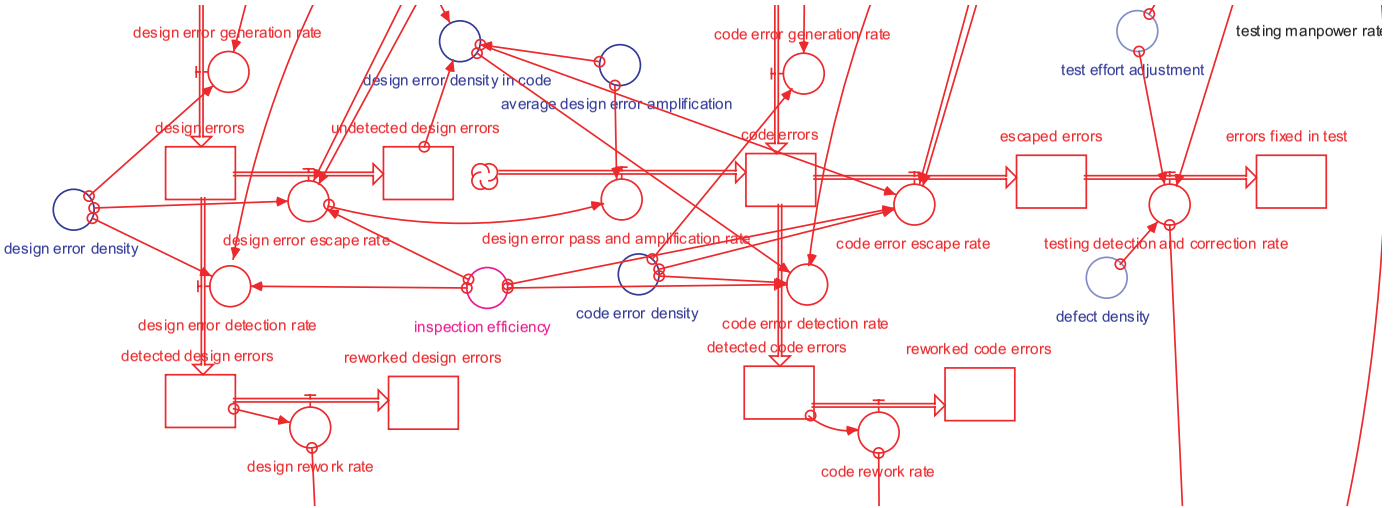


Figure 5.4. Inspection model task and error chains.

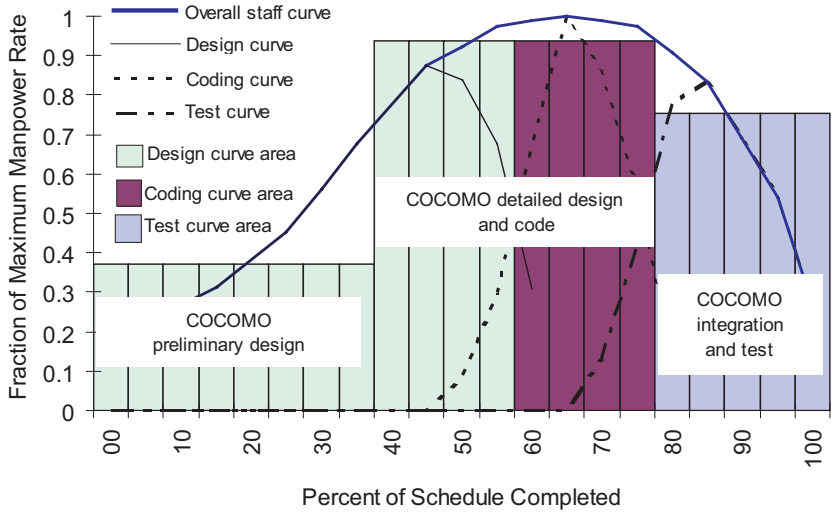


Figure 5.5. Correspondence between dynamic model and COCOMO (no inspections).

curve matches the aggregated COCOMO preliminary and detailed design, and likewise for the coding and testing curves. Unlike COCOMO, however, the staffing curves overlap between phases. The resulting staffing profiles are scaled accordingly for specific job sizes and schedule constraints.

The integration and test phase also depends on the level of errors encountered, and the curve shown is calibrated to an error rate of 50 errors/KSLOC during testing. The testing effort includes both fixed and variable components, where the variable part depends on error levels. When calibrating the model for other parameters, middle regions of published data trends were chosen as calibration points. Some parameters were set to Litton values when no other data was available.

Per the experience at Litton and accordant with other published data, inspection effort that includes preparation, meetings, and rework generally consumes approximately 5–10% of total project effort. Guidelines for inspection rates are always stated as linear with the number of pages (or lines of code), and the inspection effort is set accordingly in the model such that inspection effort does not exhibit diseconomy of scale. The default inspection effort also corresponds to a rate of 250 LOC/hour during preparation and inspection, with an average of four inspectors per meeting [Grady, Caswell 1992].

The rework effort per error is set differently for design and code. Litton data shows that the average cost to rework an error during code is about 1.3 times that for design. The cost to fix errors from [Boehm 1981] shows a relative cost for code of twice that of design. The model is set at this 2:1 ratio since it is based on a larger set of data. The resulting values are also close to the reported ranges of published data from JPL [Kelly, Sherif 1990]. The testing fixing effort per error is set at about 5 hours, which is well within the range of reported metrics. The inspection efficiency is nominally set at 60%, representing the fraction of defects found through inspections.

The utility of the model for a particular organization is dependent on calibrating it accordingly per local data. While model parameters are set with reasonable numbers to investigate inspections in general, the results using the defaults will not necessarily reflect all environments.

Parameters to calibrate for specific environments include productivity, error rates, rework effort parameters, test error fixing effort, inspection efficiency and effort, and documentation sizes.

5.3.1.2.2 Comparison to Abdel-Hamid Integrated Project Model. Using the Abdel-Hamid model as a conceptual reference [Abdel-Hamid, Madnick 1991], this model replaces the software production, quality assurance and system testing sectors. The model excludes schedule pressure effects, personnel mix, and other coding productivity determinants so as to not confound the inspection analysis.

Major departures from the Abdel-Hamid model cover manpower allocation, disaggregation of development activities, and error propagation. In his model, only generic quality-assurance activities are modeled for error detection on an aggregate of design and code. Instead of the Parkinsonian model in [Abdel-Hamid, Madnick 1991] where QA completes its periodic reviews no matter how many tasks are in the queue, resources are allocated to inspections and rework as needed. This organizational behavior is thought to be an indication of a higher maturity level. Instead of suspending or accelerating reviews when under schedule pressure, error detection activities remain under process control.

The other major structural difference is the disaggregation of development phases. The Abdel-Hamid model only has a single rate/level element that represents an aggregate of design and coding. In this formulation, phases are delineated corresponding to those in the classic waterfall model so that design and coding are modeled independently.

5.3.1.3 Model Demonstration and Evaluation

Simulation test case results were compared against collected data and other published data, existing theory, and other prediction models. Testing included examining the ability of the model to generate proper reference behavior, which consists of time histories for all model variables. Specific modeling objectives to test against include the ability to predict design and code effort with or without inspections, preparation and inspection meeting effort, rework effort, testing effort, and schedule and error levels as a function of development phase.

Approximately 80 test cases were designed for model evaluation and experimentation. The reference test case is for a job size of 32 KSLOC, an overall error injection rate of 50 errors per KSLOC split evenly between design and coding, and an error multiplication factor of unity from design to code. Refer to [Madachy 1994b] for complete documentation of the simulation output for this test case and substantiation of the reference behavior.

Other test cases were designed to investigate the following factors: use of inspections, job size, productivity, error generation rates, error multiplication, staffing pro-

files, schedule compression, use of inspections per phase, and testing effort. The reader is encouraged to read [Madachy 1994b] for complete details on the experimental results.

5.3.1.3.1 Replication of Reference Behavior. Inspection effort reference behavior used in problem definition from Michael Fagan's pioneering work [Fagan 1986] is shown in Figure 5.6. The simulation model also provides staffing profiles with and without inspections as shown in Figure 5.7, and the baseline results qualitatively match the generalized effort curves from Fagan.

The net return on investment (ROI) from inspections demonstrated in the reference behavior figures is the test effort saved minus the extra early effort for inspections (visualized as the difference between the curves in the back end minus the difference in the front end). Assessment of detailed quantities is addressed below.

5.3.1.3.2 Effects of Performing Inspections. Nine test cases were designed to evaluate the model for job size scalability, ability to be calibrated for productivity in specific environments, and to reproduce the effects of inspections.

Test case results are shown in [Madachy 1994b] consisting of cost, schedule, and error levels per phase. Predictions from COCOMO 1981 [Boehm 1981] are provided as a basis of comparison for effort and schedule. One of the salient features of the system dynamics model is the incorporation of an error model, so comparison with the COCOMO estimate provides an indication of the sensitivity of testing effort to error rates (COCOMO does not explicitly account for error rates). It is seen that the simulation results and COCOMO differ considerably for the testing and integration phases as the process incorporates inspections.

It is not meaningful to compare schedule times for the different phases between COCOMO and the dynamic model because the nonoverlapping step function profiles

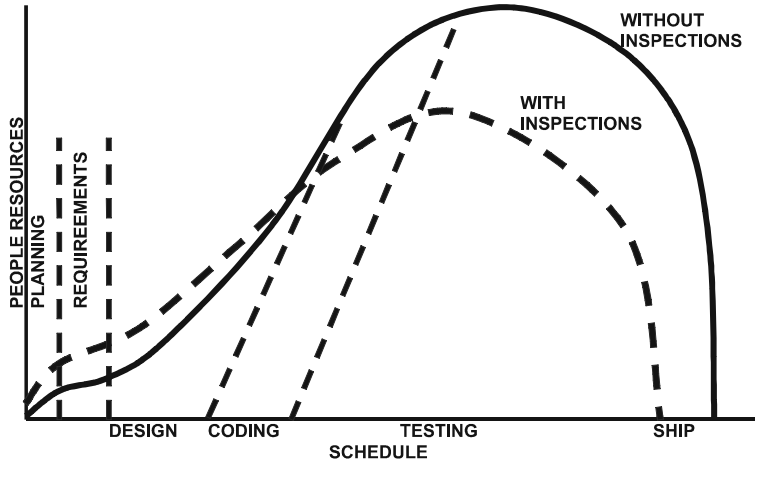


Figure 5.6. Inspection effort reference behavior [Fagan 1986].

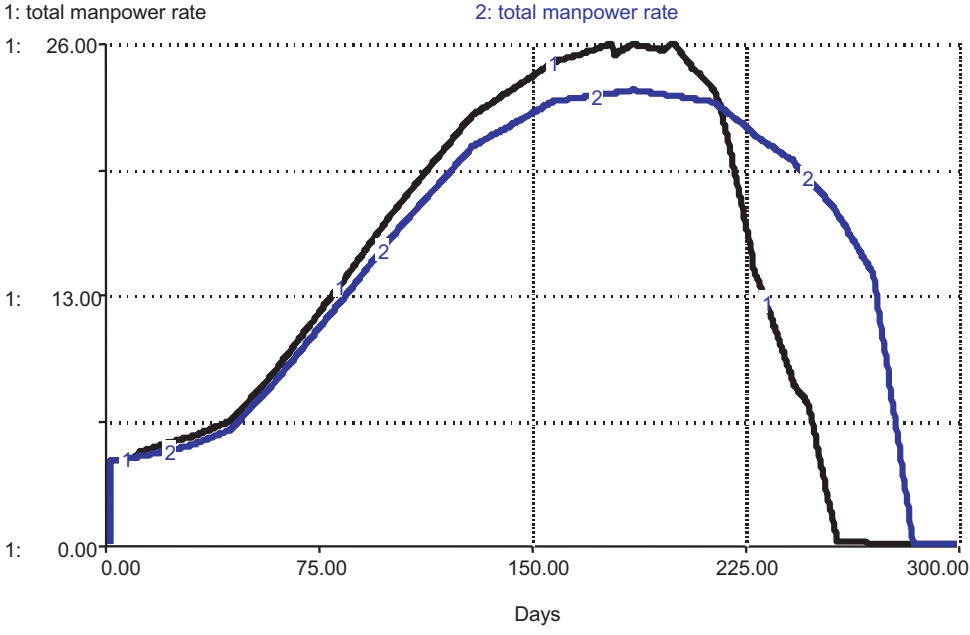


Figure 5.7. Inspection model behavior (1: with inspections, 2: without inspections).

of COCOMO will always be shorter than the overlapping staffing curves of the dynamic model as seen in Figure 5.5.

The remaining error is zero in each test case since ample resources were provided to fix all the errors in testing. If insufficient testing manpower or decreased productivity is modeled, then some errors remain latent and the resultant product is of lower quality.

Another major result is a dynamic comparison of manpower utilization for test cases with and without inspections. The curves demonstrate the extra effort during design and coding, less test effort and reduced overall schedule, much like the reference effect on project effort previously shown by Fagan [Fagan 1976].

Many effects can be gleaned from the results such as the return on investment of performing inspections, relative proportions of inspection effort, schedule performance, defect profiles, and others. It is seen that inspections add approximately 10% effort in the design and coding phases for these representative test cases, and reduce testing and integration effort by about 50%. The schedule for testing can also be brought in substantially to reduce the overall project schedule. These results are corroborated by numerous projects in the literature and verified by experts interviewed during this research.

5.3.1.3.3 Analysis of Inspection Policies. The results described above correspond to equal inspection practices in design and code, though an alternate policy is to vary the amounts. The model can be used to investigate the relative cost effectiveness of different strategies.

The model provides insight into the interplay of multiple parameters that determine the threshold at which inspections become worthwhile (i.e., the cost of fixing errors is more expensive than the inspection effort). The following sections provide some results of the multivariate analysis.

5.3.1.3.4 Error Generation Rates. The effects of varying error generation rates are shown in Figure 5.8 from the results of multiple test cases. The overall defects per KSLOC are split evenly between design and code for the data points in the figure.

It is seen that there are diminishing returns of inspections for low error rates. The breakeven point lies at about 20 defects/KSLOC for the default values of inspection effort, rework, and error fixing.

Although inspection and preparation effort stay fairly constant, the rework and test error fixing effort vary considerably with the defect rates. If there is a low defect density of inspected artifacts, then inspections take more effort per detected error and there are diminishing returns. For this reason, a project may choose not to perform inspections if error rates are already very low. Likewise, it may not always be cost effective to inspect code if the design was inspected well, as seen in [Madachy 1994b].

Though not explicitly modeled, there is a feedback effect of inspections that tends to reduce the error generation rate. As work is reviewed by peers and inspection metrics are publicized, an author becomes more careful before subjecting his work to inspection.

5.3.1.3.5 Error Multiplication. Defect amplification rates vary tremendously with the development process. For example, processes that leverage off of high-level design tools that generate code would amplify errors more than those using a design language that is mapped 1:1 with code.

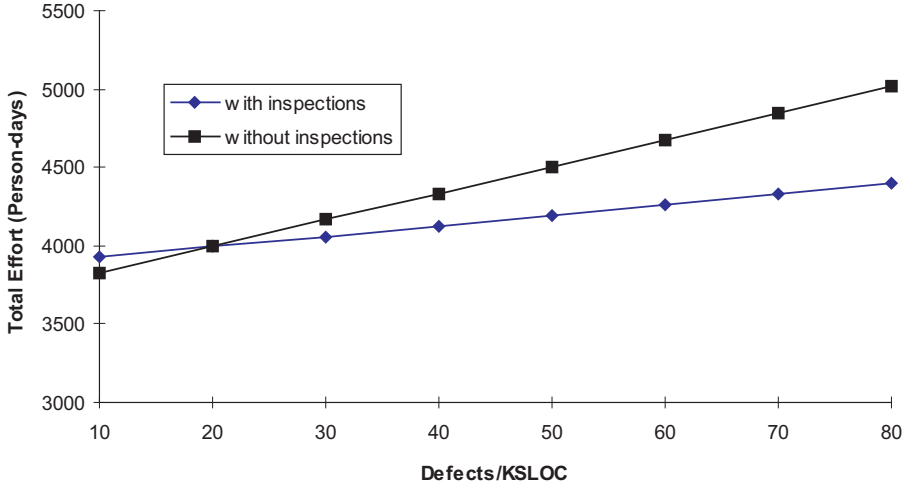


Figure 5.8. Error generation rate effects.

Figure 5.9 shows the effect of design error multiplication rate for performing inspections versus not doing so. It is seen that as the multiplication factor increases, the value of inspections increases dramatically.

The effects of error multiplication must be traded off against error injection rates for different inspected documents. If error multiplication is very high from design to code, then design inspections become relatively more advantageous through multiplication leverage (and code inspections are very important if design inspections are not well performed). Conversely, if error multiplication is low, then inspection of design documents is not as cost-effective compared to high multiplication rates.

5.3.1.3.6 DESIGN VERSUS CODE INSPECTIONS. An organization may opt to inspect a subset of artifacts. The literature suggests that design inspections are more cost-effective than code inspections. The model accounts for several factors that impact a quantitative analysis such as the filtering out of errors in design inspections before they get to code, the defect amplification from design to code, code inspection efficiency, and the cost of fixing errors in test. Experimental results show the policy trade-offs for performing design inspections, code inspections, or both for different values of error multiplication and test error fixing cost [Madachy 1994b].

5.3.1.3.7 SCHEDULE COMPRESSION. The effects of schedule compression are shown in Figure 5.10, where the relative schedule (desired/nominal) is varied from 1 to 0.7. With a reduced schedule, the average personnel level increases and the overall cumulative cost goes up nonlinearly.

5.3.1.4 Derivation of a Detailed COCOMO Cost Driver

As an example of dynamic modeling contributing to static modeling, phase-sensitive effort multipliers for a proposed cost driver, *Use of Inspections,* were experimentally

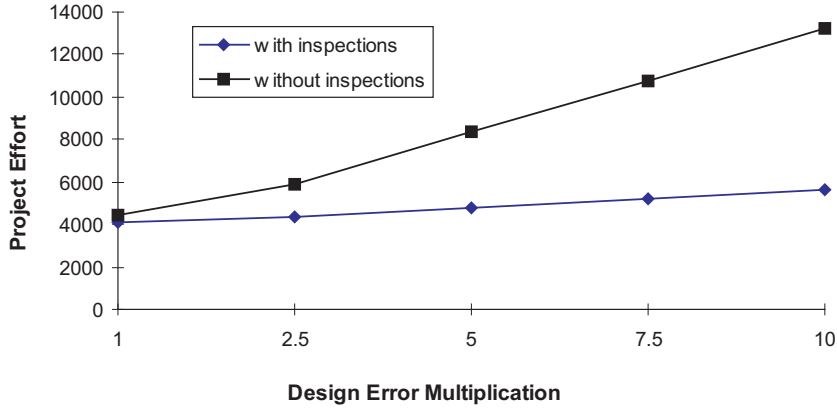


Figure 5.9. Error multiplication rate effects.

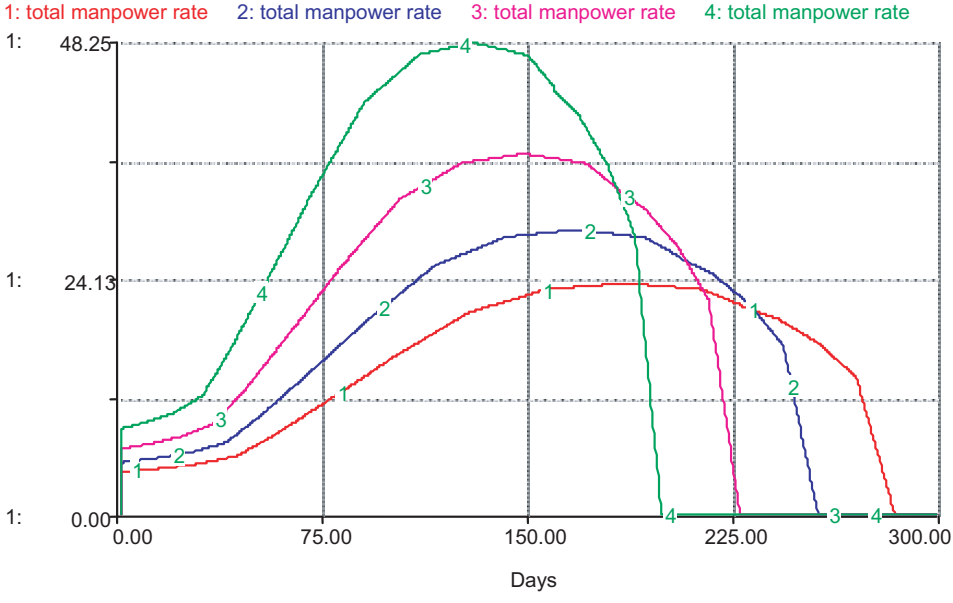


Figure 5.10. Total manpower rate curves for relative schedule (1: relative schedule = 1, 2: relative schedule = 0.9, 3: relative schedule = 0.8, 4: relative schedule = 0.7).

derived. The model parameters *design inspection practice* and *code inspection practice* were mapped into nominal, high, and very high cost driver ratings according to Table 5.2. The dynamic model was calibrated to zero inspection practice being equivalent to standard 1981 COCOMO effort and schedule. This assumption is valid because the project database used for calibrating 1981 COCOMO did not contain any projects that performed formal inspections.

Figure 5.11 shows the model-derived effort multipliers by COCOMO phase. This depiction is a composite of several test cases, and there is actually a family of curves for different error generation and multiplication rates, inspection efficiencies, and costs of fixing defects in test. The effort multiplier for high in the integration and test phase is not necessarily exactly halfway between the other points because there is a fixed amount of testing overhead, whereas half of the error fixing effort is saved.

Table 5.2. Rating guidelines for cost driver, *Use of Inspections*

Simulation Parameters		COCOMO Rating for *Use of Inspections*
design inspection practice	*code inspection practice*	
0	0	Nominal
0.5	0.5	High
1	1	Very High

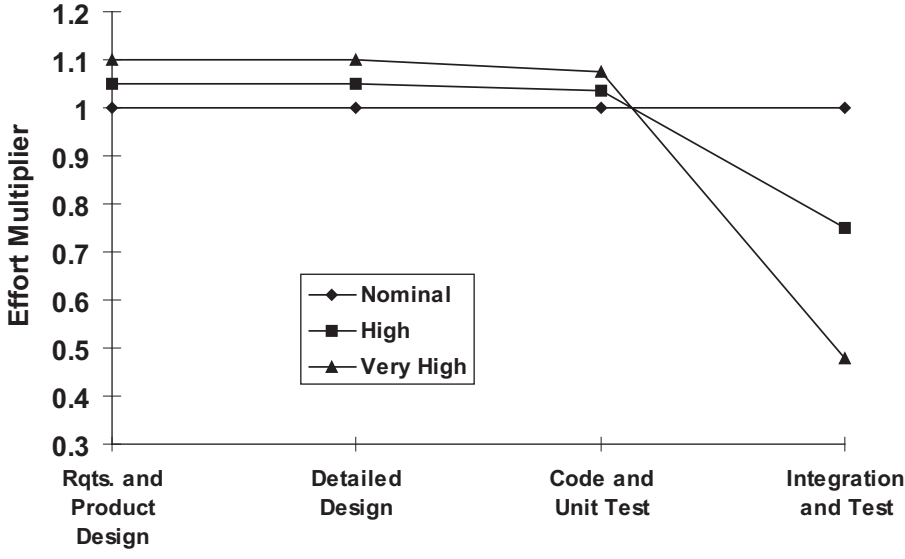


Figure 5.11. Derivation of phase-specific cost driver for *Use of Inspections.*

The results are unique in one regard because no other detailed COCOMO effort multipliers cross the nominal line, and this is the first instance of a cost driver increasing effort in some phases while reducing it in others. The dynamic model can be enhanced for additional COCOMO cost drivers, and investigation of their dynamic effects can be undertaken. Conversely, one can also derive effort multipliers for different COCOMO cost drivers by constructing unique models for investigation. For instance, the model can be used as a platform for varying levels of experience to assist in deriving multiplier weights.

5.3.1.4.1 Validation Against Industrial Data. Many comparisons of model output against specific project data were performed and are described in [Madachy 1994b]. One comparison is for return from inspections during testing, where the return is the test effort saved. The results show that the model is well-balanced to the experience at Litton. Testing effort and schedule performance for the inspection-based project at Litton is also compared against the model. Using both default model parameters and Litton-specific parameters, the model again matches actual project data well.

Another comparison afforded by available data is the ratio of rework effort to preparation and meeting inspection effort. This is done for over 500 data points from Litton and JPL data [Kelly, Sherif 1990]. The model predicts the average amount of rework effort to overall inspection effort for both organizations within 9%. A detailed validation summary of the model, including all time history outputs, is provided in [Madachy 1994b].

5.3.1.5 Conclusions

The model successfully demonstrated several phenomena of software projects, including the effects of inspections and management decision policies. It was also shown that the model is scalable for project size and calibratable for productivity, error generation, and error detection.

The model endogenously illustrates that under nominal conditions, performing inspections slightly increases up-front development effort and returns more in decreased testing effort and schedule. However, the cost-effectiveness of inspections depends on phase error injection rates, error amplification, testing error fixing effort, and inspection efficiency.

Complementary features of dynamic and static models for the software process were also shown. Static models were used to nominally calibrate the dynamic model, and the dynamic model improved on static assumptions. There are many interesting problems and challenging opportunities for extensions of the research. A detailed outline of future directions is provided in [Madachy 1994b].

5.3.1.6 Modification for Walk-Throughs

The inspection model was modified at the Guidance and Control Systems division of Litton to model other types of peer reviews, particularly walk-throughs.

Local walk-through data was used to parameterize the model and scenarios were run to quantify the expected cost and schedule impacts of peer reviews. The experimentation helped to solidify plans for peer reviews on specific projects. On others, it was shown that performing additional formal peer reviews may not be worthwhile. The model has also helped to educate executives and developers alike on the importance of peer reviews, and provided motivation to optimize the peer review processes.

The model is general enough that it required no structural changes to handle other peer review types besides inspections. The following calibration parameters were determined for each project based on existing local baseline data as refined for specific projects:

- Nominal productivity
- Review efficiency
- Design defect density
- Code defect density
- Average design defect amplification

Project-specific parameters include

- Job size
- COCOMO effort parameter
- Schedule constraint

The parameters below represent management decisions regarding scheduled processes for each project:

- Design walk-through practice (percent of design packages reviewed)
- Code walk-through practice (percent of code reviewed)

5.3.2 Example: Inspection Process Data Calibration

Example datasets of inspection data are shown here with respect to model parameter calibration. Table 5.3 shows sample summary data aggregated from inspection data records for each review held on a project. An overall inspection efficiency of 60% on the project denotes that 60% of all defects where found during inspections, and the other defects were found during integration and system testing. The inspection efficiencies directly translate into the model parameters for efficiency (yield %), representing the defect filters. The 60% from this dataset was used as a baseline in the inspection model.

The total inspection effort for this project is distributed as follows: preparation effort, 27%; inspection meeting effort, 39%; and rework effort, 34%. This type of data can also be used to parameterize the model with respect to effort proportions for the life-cycle activities.

An illustration of data used to calibrate the defect introduction rates, detection rates, and review efficiencies for the peer review model in another installation is shown in Figure 5.12 (the data has been sanitized from some actual project data). In this case, defect data came from multiple defect-finding activities corresponding to different

Table 5.3. Example inspection summary statistics

Subject Type	Number of Inspections	Total Defects	Total Majors	Inspection Effort (person-hours)	# Pages	LOC	Defects/ Page	Defect Removal Effectiveness (major defects/ person-hour)
Requirements Description (R0)	21	1243	89	328	552	0	2.25	0.272
Requirements Analysis (R1)	32	2165	177	769	1065	0	2.03	0.230
High Level Design (I0)	41	2398	197	1097	1652	0	1.45	0.180
Low Level Design (I1)	32	1773	131	955	1423	28254	1.25	0.137
Code (I2)	150	7165	772	4612	5047	276422	1.42	0.167
Test Procedure (IT2)	18	1495	121	457	1621	0	0.92	0.265
Change Request	24	824	27	472	1579	340	0.52	0.057
Other	2	57	4	27	31	781	1.84	0.150
Grand total	320	17120	1518	8716	12970	305797	1.32	0.174

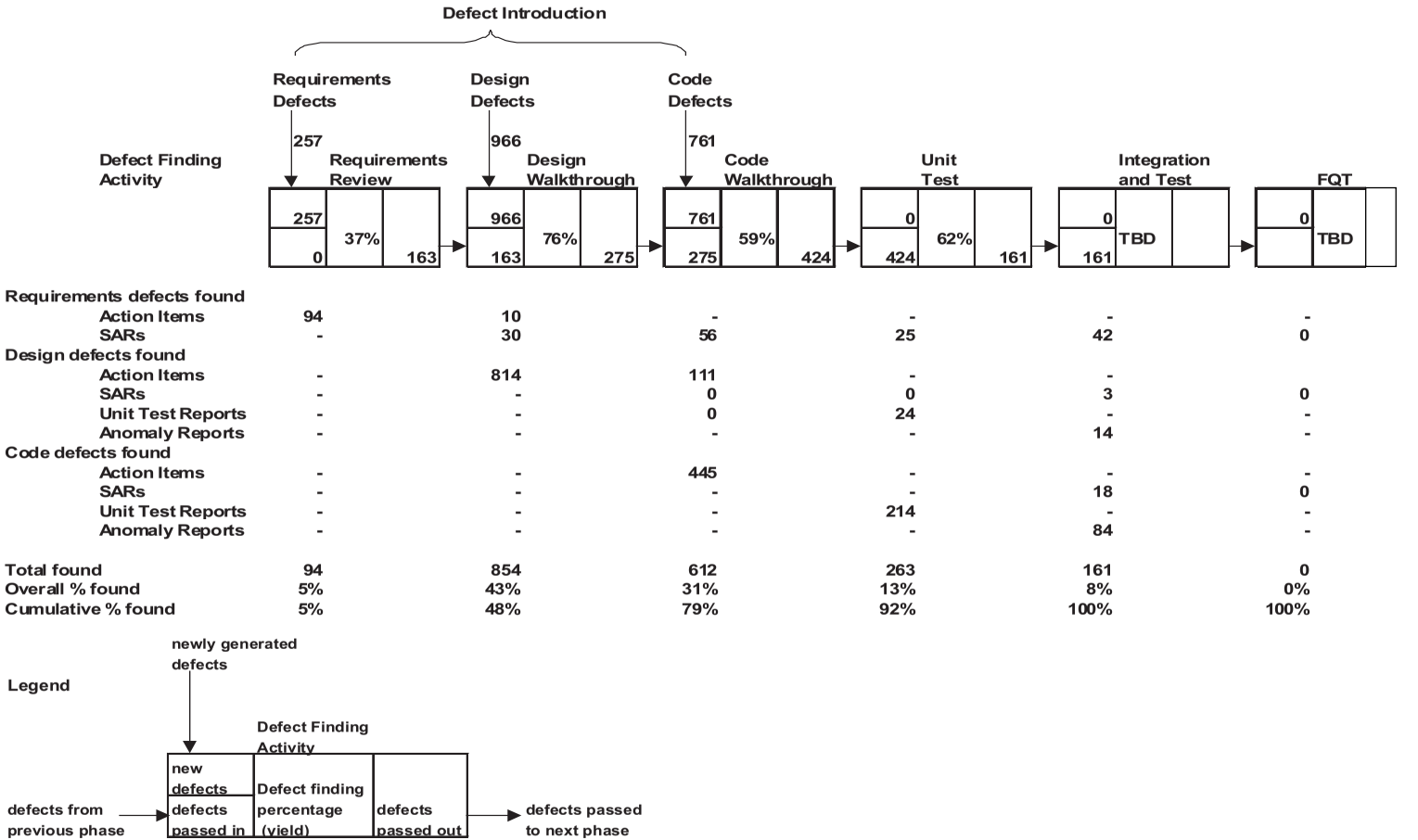


Figure 5.12. Sample defect calibration data.

stages of the process. Data was collected during the defect-finding activities across the top using the reporting mechanisms listed in the left column (an SAR is a software action request), and the review efficiencies are calculated accordingly.

From this type of dataset, a defect flow chain could be calibrated in terms of defect introduction rates and defect detection rates at different junctures of the chain. According to the model structuring principles in Chapter 3, a level would be associated with each step of the process and detection filters would be placed in between them. The introduction rates would be normalized for the project size. All the other values are readily available in the format provided by Figure 5.12.

5.4 GLOBAL PROCESS FEEDBACK (SOFTWARE EVOLUTION)

Global process feedback refers to feedback phenomena associated with the evolution of a fielded software product as it is maintained over time. Software evolution is an iterative process that implements development, adaptation, enhancement, and extension due to changes in application, business, organizational, and technological environments. Changes are initiated, driven, controlled, and evaluated by feedback interactions. Feedback also plays a large role in related organizational business processes.

Why is it natural for global software process feedback and software evolution to occur? A software product encapsulates a model of the real world, but the real world keeps changing. Thus, the rapid pace of change also tends to accelerate software evolution. A need for continual enhancement of functional power is one of the inevitable pressures that emerge when evolving software that addresses real-world problems or automates real-world activities [Lehman, Belady 1985].

Whereas most software process simulation addresses development projects from the beginning through initial delivery, work in the area of software evolution focuses on the entire evolution process and long-term issues associated with feedback from all sources.

Achieving controlled and effective management of such a complex multiloop feedback system and improving its reliability and effectiveness is a major challenge. The problems will become more challenging given current shifts to COTS-intensive systems; distributed development; outsourcing; open-source, multisupplier environments, and so on. Thus work in global feedback is well suited to addressing these new and emerging areas.

Manny Lehman and colleagues have been at the forefront of global feedback modeling for many years [Lehman, Belady 1985, Lehman 1996, Lehman 1998, Wernick, Lehman 1999, Chatters et al. 2000, Lehman, Ramil 2002]. Lehman first observed empirically in 1971 the strong presence of external controlling feedback in the software evolution process [Lehman 1980]. When modeling a software process to assess its performance and improve it, feedback loops and controls that impact process characteristics as observed from outside the process must be included. The process must be modeled at a high enough level such that influences and control aspects of the process, management, marketing, users, and technical personnel can be included.

The earliest studies of evolution and feedback in the software process originated in the 1970s [Riordan 1977], but wide interest in software evolution emerged with formulation of the Feedback, Evolution And Software Technology (FEAST) hypothesis in 1993. The hypothesis of the FEAST project is that the global software process is a complex feedback system, and that major improvement will not be achieved until the process is treated as such. More information can be found at [FEAST 2001]. Per [Lehman 1998],

> It is universal experience that software requires continuing maintenance to, for example, adapt it to changing usage, organizational and technological domains. Such evolution is essential to maintain it satisfactory. Feedback and its management play a major role in driving and controlling this process. The two phenomena are strongly related. The feedback paradigm offers a powerful abstraction whose full potential and the impact it can make remain to be fully investigated.

Because of the very nature of the problem, such investigation requires widespread interdisciplinary collaboration with both industrial and academic involvement. Long-term goals must include the design and management of feedback mechanisms and control and the development and validation of a theoretical framework.

The investigation of software evolution includes the complementary concerns of *how, what,* and *why. How* addresses the achievement and control of software evolution and is considered a verb. *What* and *why* focus on the question of why software evolution occurs, its attributes, characteristics, role, and impact.

A simple example of feedback that contributes to software evolution is when users realize an existing software capability can be improved to solve another problem. Users become empowered when they understand what the software can do for them, and naturally they want more. A similar thing happens with user interfaces. Users will interpret some aspects of an interface design differently than what was intended by the designer and will desire new functionality. This is related to the requirements phenomenon that users do not know what they want until they see it, which is called I'll Know It when I See It (IKIWISI).

One of the earliest efforts applying system dynamics to global feedback processes was [Wernick, Lehman 1999], where good results were obtained from a macro model of global feedback. The model was validated against long-term data from many releases of a large software system over time.

The main structure in their model is reproduced in Figure 5.13. It simulates the software production process, the field testing of the implementation, and changes made to the specification as a result of field trials. The implementation and initial testing is represented as a third-order delay. Then there are feedback paths in the process before field trials. It is found that some units cannot be implemented, so they are eliminated and may be replaced with new or changed equivalents. The completion of a unit may also result in more specification changes.

Other feedback paths occur during field trials. It is found that some specifications need to be re-examined and possibly modified. Others are defective or unnecessary and must be removed from the system. New specifications may also be identified. The

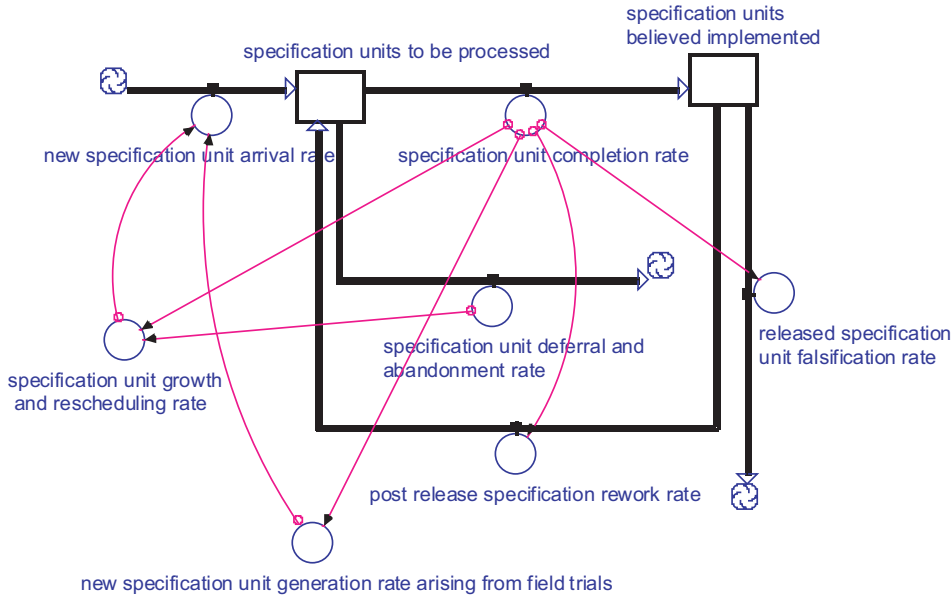


Figure 5.13. Project model with global feedback.

sequence of time trials is simulated as a continuous process, where the interval between trials is progressively reduced in the feedback paths to developers and their completing the work. The model also reflects exogenous changes made to the specifications during the project.

This relatively simple structure only needs to be parameterized with limited, early delivery data from an actual project. A comparison of simulated effort from the model versus actual effort expended on an evolution project is shown in Figure 5.14. It is seen that the model reproduces the actual effort profile well over the long time horizon. Since the model outputs closely simulate actual metrics, it was concluded from this study that external feedback to a software process may significantly influence the process. See [Wernick, Lehman 1999] for further details on the model testing and validation.

This early modeling work in global feedback helped pave the way for many follow-on research studies. The following is a more recent result in global feedback modeling on balancing work activities for optimal evolution from [Ramil et al. 2005].

5.4.1 Example: Software Evolution Progressive and Antiregressive Work

This example presents highlights from [Ramil et al. 2005]. It describes a system dynamics model used to plan and manage long-term software evolution by balancing the allocation of resources to different activities. See the original work for more details and references. Table 5.4 is a high-level summary of the model.

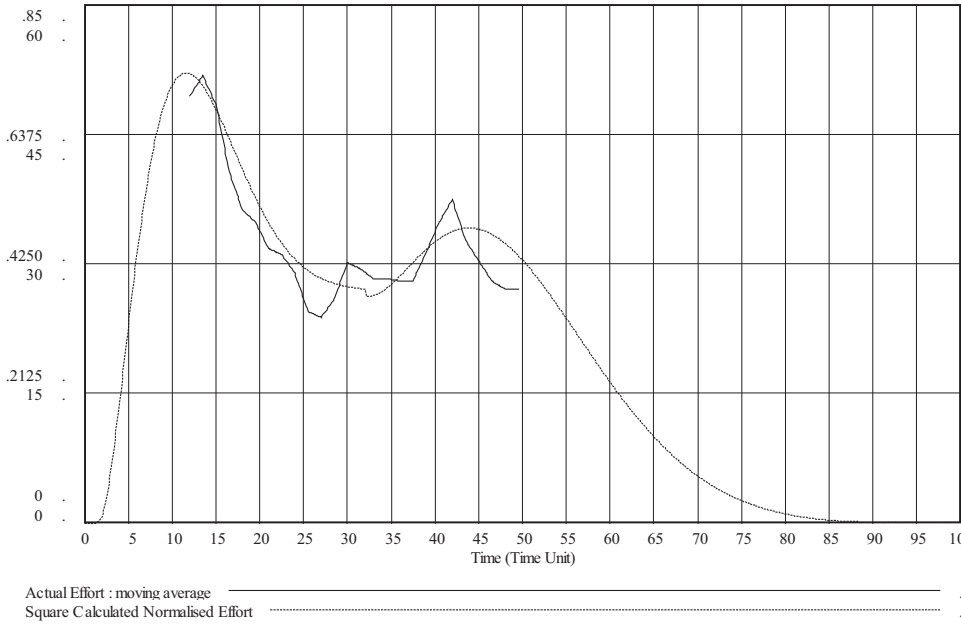


Figure 5.14. Global feedback model results: actual versus simulated effort.

Table 5.4. Software evolution progressive and antiregressive model overview

Purpose: Planning, Strategic Management, Process Improvement
Scope: Long-term Product Evolution

Inputs and Parameters	Levels	Major Relationships	Outputs
• Percent antiregressive work • Staffing policy • Release policy • Domain factors • Baseline productivity	• Work Awaiting assignment Implemented Validated components Outputs Validated elements Standby Completed Released	• Growth productivity • Impact of anti-regressive deficit	• Software modules • Effort

5.4.1.1 Introduction and Background

Software that is regularly used for real-world problem solving or addressing a real-world application must be continually adapted and enhanced to maintain its fitness to an ever changing real world, its applications, and application domains. This type of activity is termed *progressive.* As evolution continues, the complexity (functional, structural) of the evolving system is likely to increase unless work, termed *antiregressive,* is undertaken to control and even reduce it. However, as progressive and antiregressive work compete for the same pool of resources, management has to estimate the amount of resources applied to each. A systems dynamics model is described that can support decision making regarding the optimal personnel allocation over the system lifetime.

Real-world software must be progressively fixed, adapted, and enhanced, that is, *evolved,* if it is to remain satisfactory to its stakeholders, as evidenced by the universally experienced need for continual software maintenance [Lehman, Belady 1985].

The resources available over some predetermined period or for the development of a new release will be primarily intended for *progressive* activity. This represents activity that adds functionality to the system, enhances performance, and, in general, adds capability to the system as perceived by users and by marketing organizations.

As successive versions of a real-world software system emerge over its long-term evolution, source code is augmented; system size increases and fixes; adaptations, functional, and nonfunctional enhancements get implemented, which are ever more remote from the original conception. The consequence of all these changes and superposition of changes is that the software system *complexity* is likely to increase as the system is evolved. This may bring with it a decline in the *functional growth rate,* as observed in plots of system growth over releases; see, for example, [FEAST 2001]. If this issue is not promptly recognized and addressed, it is likely to lead to decreasing evolvability, increasing maintenance and evolution costs, and even stagnation.

The satisfaction of new or changed needs must not conflict with the need to ensure that the software remains evolvable. The latter is achieved by executing activities such as, for example, restructuring, refactoring, and documentation, termed, in general, *antiregressive* activities. They neutralize or reverse system degradation due to growing structural, functional, and operational complexity.

Antiregressive work is generally not regarded as of high priority in light of other pressures, but a resource trade-off has to be made. A system will degrade if focus is put only on progressive work, but if all activities are antiregressive, then system evolution comes to a halt. Between these limiting situations there must be a division of progressive and antiregressive levels of investment that achieves the best balance between immediate added functional capability and system evolution potential and longevity.

5.4.1.1.1 Model Overview. The model is driven by the observation that each unit of progressive work requires a minimal amount of antiregressive work to forestall accumulation of an antiregressive deficit [Riordan 1977]. As the required but neglected antiregressive effort accumulates over time, its impact on productivity begins to be noticeable. Only restoration to an adequate level can reverse the growth trend and restore productivity. The model provides a tool to determine what is adequate.

The model is shown in Figure 5.15. The outer loops of flows and stocks show the arrival and implementation or rejection of work requests, their validation, and delivery of the product to the field. It is visualized as a process that addresses a continuing flow of work in the form of changes in requirements, functional adaptation, enhancements, and so on.

The inner feedback loop on the bottom is a provision for delaying output of the validation step and for the authorization of rework. Some part of the output will be held, rejected, or recycled.

Parameters for new functions include productivity and staff size to include the assignment of resources to progressive work. To complete the model, there is a subgraph for splitting the effort between progressive and antiregressive work. Antiregressive work is captured as a flow rate. The expressions included in the executable model in the Vensim language are available upon request from the authors.

5.4.1.1.2 Calibration and Experimentation. A calibration process was followed, though not every measure was directly available and attribute surrogates were sometimes use [Ramil et al. 2005]. Nevertheless, the relatively simple model closely approximates real-world patterns of behavior. Figure 5.16 shows how closely the model reproduces the growth trend of an information system over 176 months of its lifetime. As illustrated, the model is able to replicate actual trends even though only a small subset of the model parameters were known. Despite possible refinements to recalibrate for specific processes, the model exemplifies the approach and is sufficient to perform some virtual experiments described next.

One set of experiments concerns the long-term consequences of different levels of antiregressive activity on system growth rate and productivity. Several values of antiregressive work are simulated, expressed in percentage of total human resources available for the evolution of the system. Figure 5.17 and Figure 5.18 permit visualization of the effect of different antiregressive policies.

Figure 5.17 shows several simulated trajectories resulting from alternative fixed-allocation strategies. For the lowest level of antiregressive work, one achieves the highest *initial* accomplishment rates, suggesting initially low antiregressive activity, to maximize initial growth rate.

Figure 5.18 indicates the impact of different levels of fixed antiregressive work on growth–productivity, the number of elements created per unit of total effort applied (where both progressive and antiregressive work are added).

Together, the two experiments suggest that a constant level, in this case approximately 60% of resources allocated to antiregressive work, maximizes long-term growth capability. This number will vary from process to process, but the important lesson is that antiregressive work in excess of some level constitutes resource wastage.

Whether restructuring occurs or not, the experiments suggest that as a system ages one may seek to maintain system growth rate or, equivalently, minimize total effort required to achieve the desired evolution rate and, hence, productivity by adjusting the level of antiregressive activity.

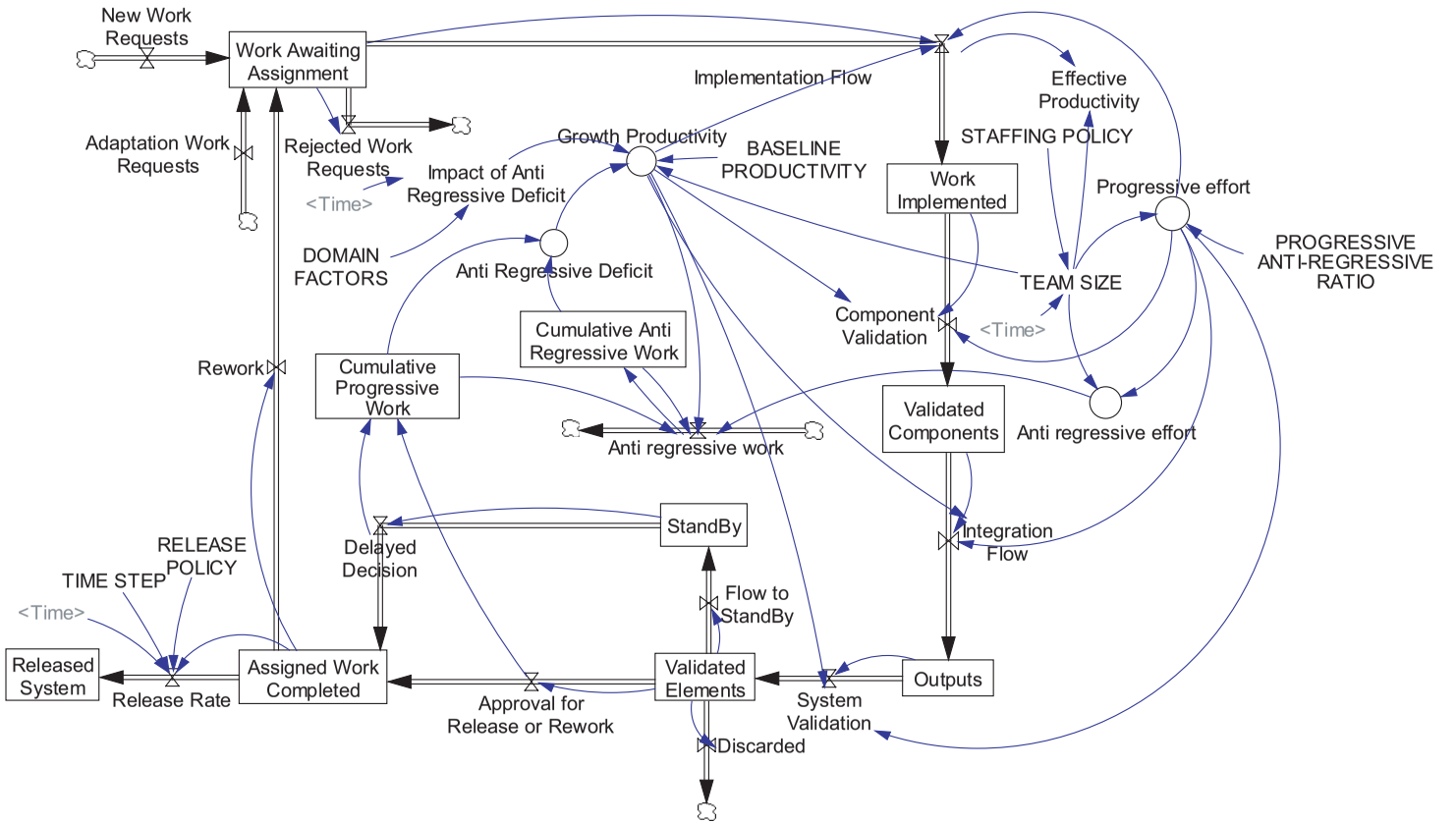


Figure 5.15. Software evolution progressive and antiregressive work model diagram.

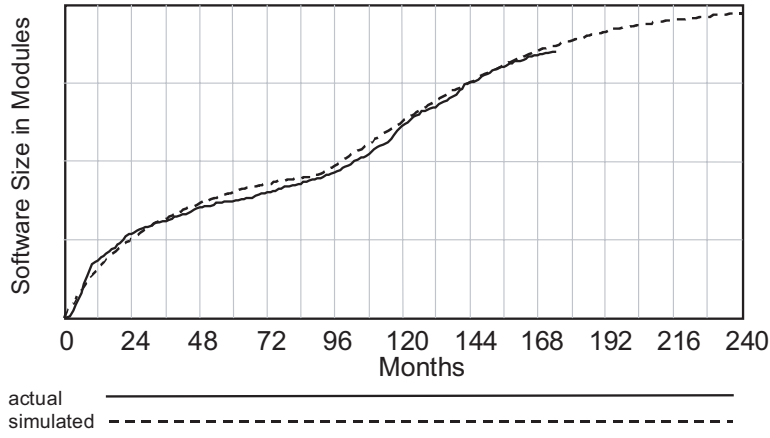


Figure 5.16. Simulated model output versus actual growth trend for an information system.

5.4.1.1.3 Further Work and Conclusions. The optimal level of antiregressive work is likely to vary from process to process and over the operational life of a long-lived software system. Eventually, the desirable level of antiregressive effort will stabilize in a way that still permits the allocation of sufficient resources to ensure further effective system evolution. This aspect may be considered for further refinements.

More detailed policies and mechanisms such as the inclusion of a management feedback control loop that changes the degree of antiregressive activity over time in response to some simulated circumstance can be easily added to explore their impacts.

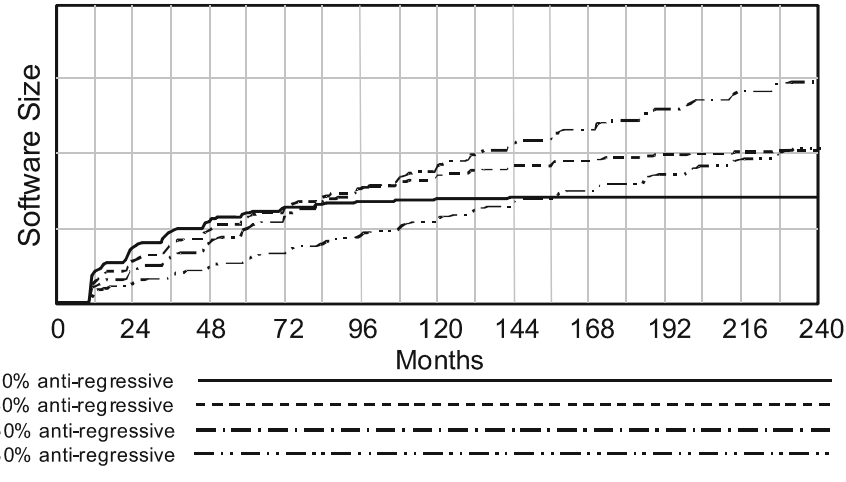


Figure 5.17. Growth trends under different levels of antiregressive work.

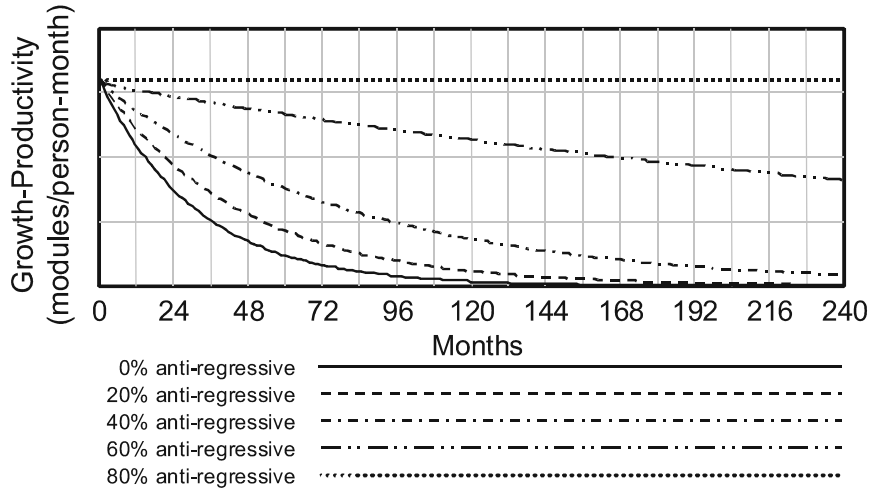


Figure 5.18. Growth–productivity under different policies.

For wider application, the model needs to be refined to accommodate for allocation of work to a wider variety of concerns than antiregressive and progressive work. Extension to more general paradigms, such as component-based and reuse-based processes, and customization of the model to specific process instances is likely to raise issues. These would likely include the need for measures of stakeholder satisfaction and system value.

The described model can serve as the core of a tool to support decisions regarding allocation of personnel to evolution activities over the application lifetime. Process modeling at an aggregated, high level of abstraction can serve as the basis for a tool to assist managers to recognize and control the various influences on long-term behavior. By taking these into account, they may can manage and control complexity to maintain the rate of system evolution at the desired or required level.

Control and mastery of system evolution is vital in a society increasingly reliant on aging software in which increased size, more interdependent functionality, larger numbers of integrated components, more control mechanisms, and a higher level of organizational interdependency are likely to lead to a decrease in evolvability. As society relies increasingly on software, planning and management of complex, dynamic, and ever more widespread and integrated evolution processes is becoming increasingly critical.

5.5 SOFTWARE REUSE

Software reuse was introduced in Chapter 1 under life-cycle models, and will not be repeated here. There are many variants of reuse. Other application areas like COTS, product line strategies (see Chapter 6), and open-source development all employ reuse

in some form or another. Virtually all software artifacts can be reused in these contexts, not only source code.

For illustrative purposes, a simple model of reuse is shown in Figure 5.19 using standard structures from Chapter 3. It contains two flows into the completed software: one for newly developed software and one for modified (reused) software. Note that there are different rates associated with the new software development and the modification of existing software. There is also a separate level for the amount of existing software being modified.

The effort for reusing software may be more or less expensive than for new software depending on the nature of the reuse and state of the existing software. The adaptation factor is a fraction representing the relative effort needed to adapt software compared to if it were new. A value of 20% means that one-fifth of the effort is required to modify the software relative to if it were developed new.

The nonlinear COCOMO II reuse model can be employed for static calculations of the relative effort to reuse software, and can serve as a jumping point for a dynamic model elaboration. The reuse parameters in the COCOMO II model provide visibility into some of the important factors and tradeoffs:

- Percent of design modified
- Percent of code modified
- Percent of reintegration required
- Amount of assessment and assimilation

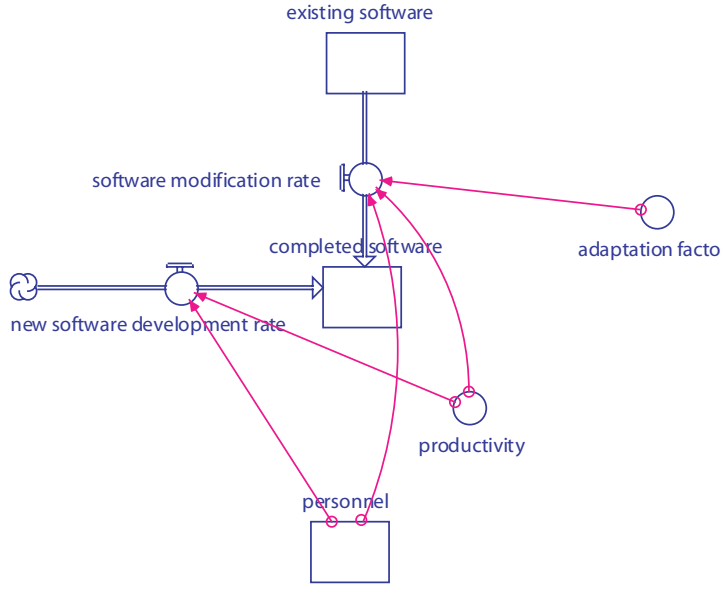


Figure 5.19. Top-level reuse model.

- Degree of software understanding
- Programmer familiarity

These reuse parameters are used to derive an overall adjustment factor. See the COCOMO II summary appendix for more details of these definitions and the calculations for relative effort.

The resulting effort from a static model like COCOMO can be transformed into a reuse rate. Moreover, the COCOMO II reuse parameters are static approximations that can be reformulated into a dynamic model accounting for feedback interactions. For example, the degree of software understanding is not a static value, but would follow from the amount and type of changes incurred to the software. The following example is a modeling study of reuse and fourth-generation languages that employs a dynamic model of reuse.

5.5.1 Example: Reuse and Fourth-Generation Languages

A USC research study investigated the dynamic effects of reuse and fourth-generation languages (4GLs) [Lo 1999]. The purpose of the modeling project was to study the effects of reuse and very high-level languages (RVHL) in a RAD context for reducing schedule. Using different kinds of programming languages and levels of reuse may affect the schedule and effort, so understanding their effects on the software process will benefit management planning strategy. Table 5.5 is a high-level summary of the model.

5.5.1.1 Background

Different languages have different effort impacts on portions of the development process, such as design, coding, and testing. High-level programming languages typi-

Table 5.5. Reuse and fourth-generation languages model overview

Purpose: Process Improvement, Planning
Scope: Development Project

Inputs and Parameters	Levels	Major Relationships	Outputs
• Language level • Project size • Personnel • Hiring delay • Attrition rate • Reuse percentage • Reuse components • Decay rate • Nominal productivity • Learning curve switch • Incompressibility factor	• Personnel • Tasks Required Designed Designed with errors Coded Coded with errors Completed	• Learning curve • Task approval • Communication overhead • Code adoption rate • Productivity	• Staffing per activity • Effort per activity • Schedule per activity • Reuse level

cally come with source code libraries, which make code generation and testing more efficient. A higher-level programming language provides similar advantages to reuse because it does automatic code generation. It "reuses" packaged code.

Different language levels affect the error level in each phase because the richer the language, the more you have to test or verify as you build. However, the richer the language, the more training is involved. Reusing assets also reduces human effort and increases productivity in each phase.

5.5.1.2 Model Description

There are four phases represented in the model: requirements, design, coding, and approval phases. By default, each phase uses COCOMO phase effort distributions (see [Lo 1999] for the values). Language effects on effort were also quantified. Effort ratios of 1:6.7 for 3GL to 4GL and a ratio of 1:3 for Assembly to 3GL were used according to [Verner, Tate 1988].

Interviews with language level and reuse experts suggested that high-level languages require more training. A learning curve is used to represent this, with a 90% learning curve for 3GL and an 80% learning curve for 4GL. Both log-linear and DeJong learning curves are used in the model. A DeJong curve models processes in which a portion of the process cannot improve [Raccoon 1996]. Per interviews, different error rates for the different levels are applied in the design and code phases according to the language level being used.

A reuse structure is in the model that reduces the size of flows into the software development phases. This reduces effort depending on the percentage of reuse on the project. A level for reuse code was added, which can monitor the actual reuse level during the project development. But a reuse policy is not only for code reuse. It also includes pattern reuse, design reuse, test case reuse, and so on. In each case of reuse, there can be different degrees of levels and factors. This model treats reuse as if it reduces the entire software size.

A model of software production provides the personnel staffing profile. A communication overhead factor is also included that reduces productivity. With this, adding an infinite number of people will not finish the project in zero time.

Figure 5.20 presents a causal loop diagram for the model. The causal diagram shows that the language level affects defect density, productivity, and learning curve percentages. Increased language level will decrease defect density and fewer errors occur. Higher language level also increases the productivity. But increased language level will increase the learning curve percentages, which results in a slower learning rate. By increasing the number of human resources, it increases communication overhead and decreases productivity.

With less defects, a certain amount of people and high productivity will increase the design, code, and approved tasks rates. Higher productivity also increases code adoption and increases code reuse. Since design, coding, task approval, and code reuse are all increased, it is more efficient to complete tasks. A higher completed task rate reduces the schedule time.

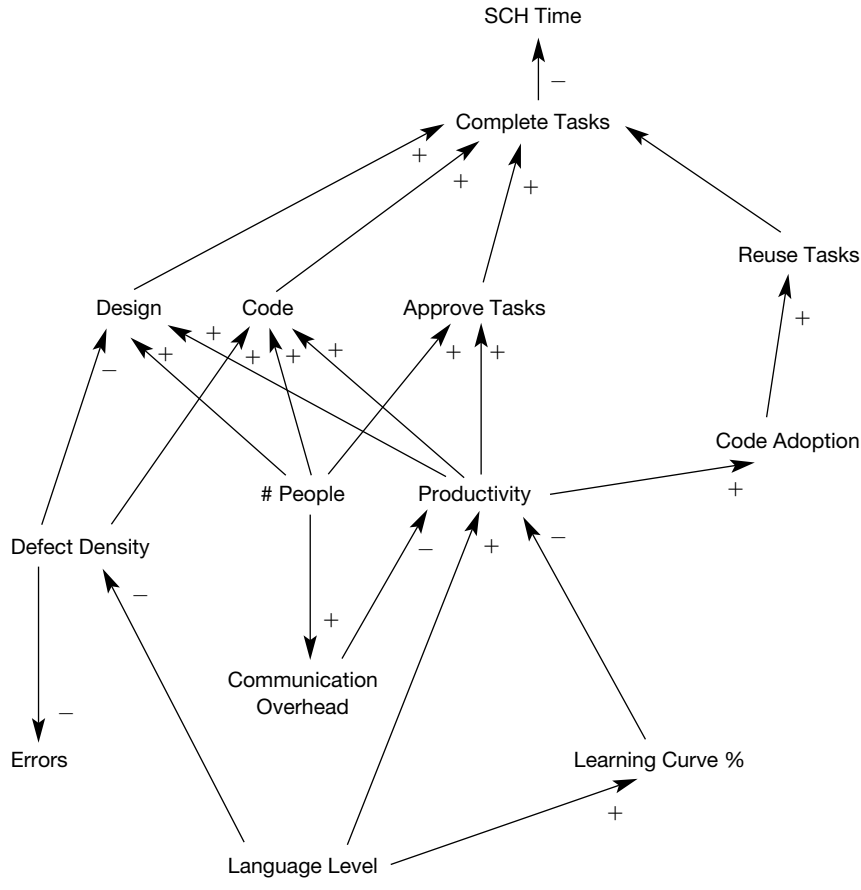


Figure 5.20. Reuse and fourth-generation languages causal loop diagram.

The primary components in the executable model are shown in Figure 5.21. Detailed model components are in Figure 5.22, Figure 5.23, Figure 5.24, and Figure 5.25 with descriptions following the figures.

In the software production structure, development starts with a project-size level and flows into requirements, design, code, and approve phases to complete tasks. The flow rates for each phase are determined by

$$\text{communication overhead} \cdot \text{personnel} \cdot \text{productivity/phase effort \%}$$

There is error detection used in design and code phases, and rework flows back into the product line. The error rate depends on the defect density, which is controlled by language level.

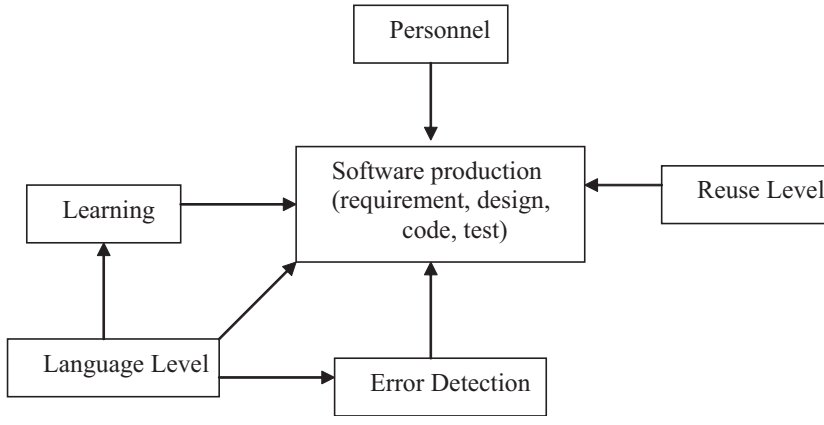


Figure 5.21. Reuse and fourth-generation languages model components.

The reuse structure reduces the project size when there is any reuse percentage. The reuse components act like a library to provide reusable assets. The reuse components have a decay rate due to the change of needs or old components. The adoption rate adjusts reuse tasks and is formulated as

$$\text{personnel} \cdot \text{productivity} \cdot \text{learning curve} \cdot \text{reuse percentage/adoption factor}$$

The reuse level is used to check the actual reuse in the project where reuse level equals reuse tasks/complete tasks · 100%.

The language level controls the language factor that modifies nominal productivity. The language level also affects the required effort, which is used to calculate the effort for each phase. There are two learning curves used that require a learning percentage, which is also controlled by language level. The original paper [Lo 1999] has more details on the learning curve formulas and parameters.

The personnel staffing component defines the human resources for the project. The hiring rate is affected by a hiring delay, and the attrition rate represents personnel lost. Adding more people will increase communication overhead and may decrease the software development rate.

5.5.1.3 Sample Results and Conclusions

Several types of tests were performed to assess the model and draw phenomenological conclusions. Scenarios were run to verify a nominal case, to test learning curves, the effects of language levels, the effects of varying degrees of reuse, personnel levels, and combined strategies. The detailed inputs and outputs are provided in the original report in [Lo 1999].

Results for varying reuse are shown in Figure 5.26. Reuse will reduce the effective project size, so the effort in all phases and the schedule time are all reduced. A test of language levels that compared 3GL to 4GL with everything else constant is shown in Figure 5.27; 4GL finished first.

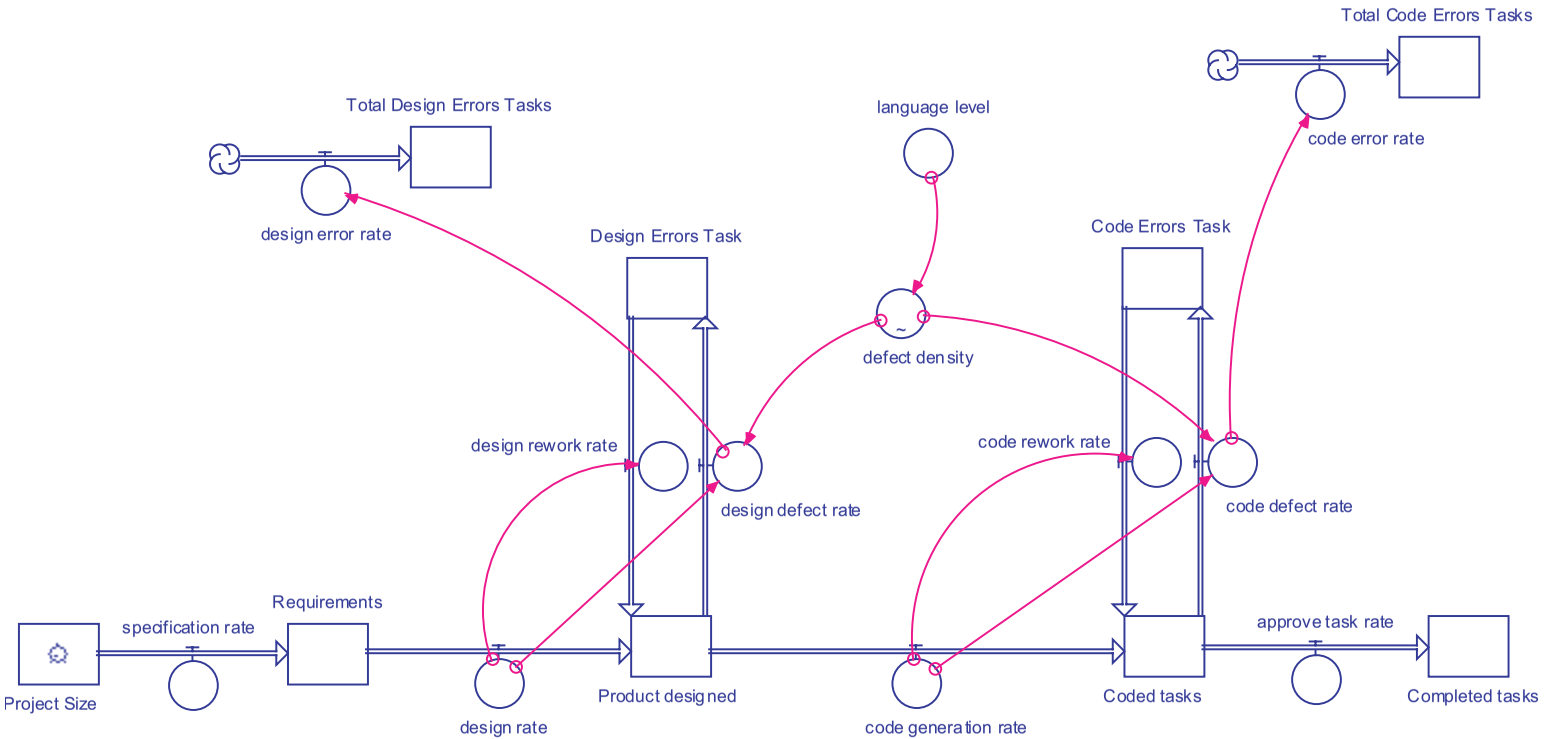


Figure 5.22. Software production phases.

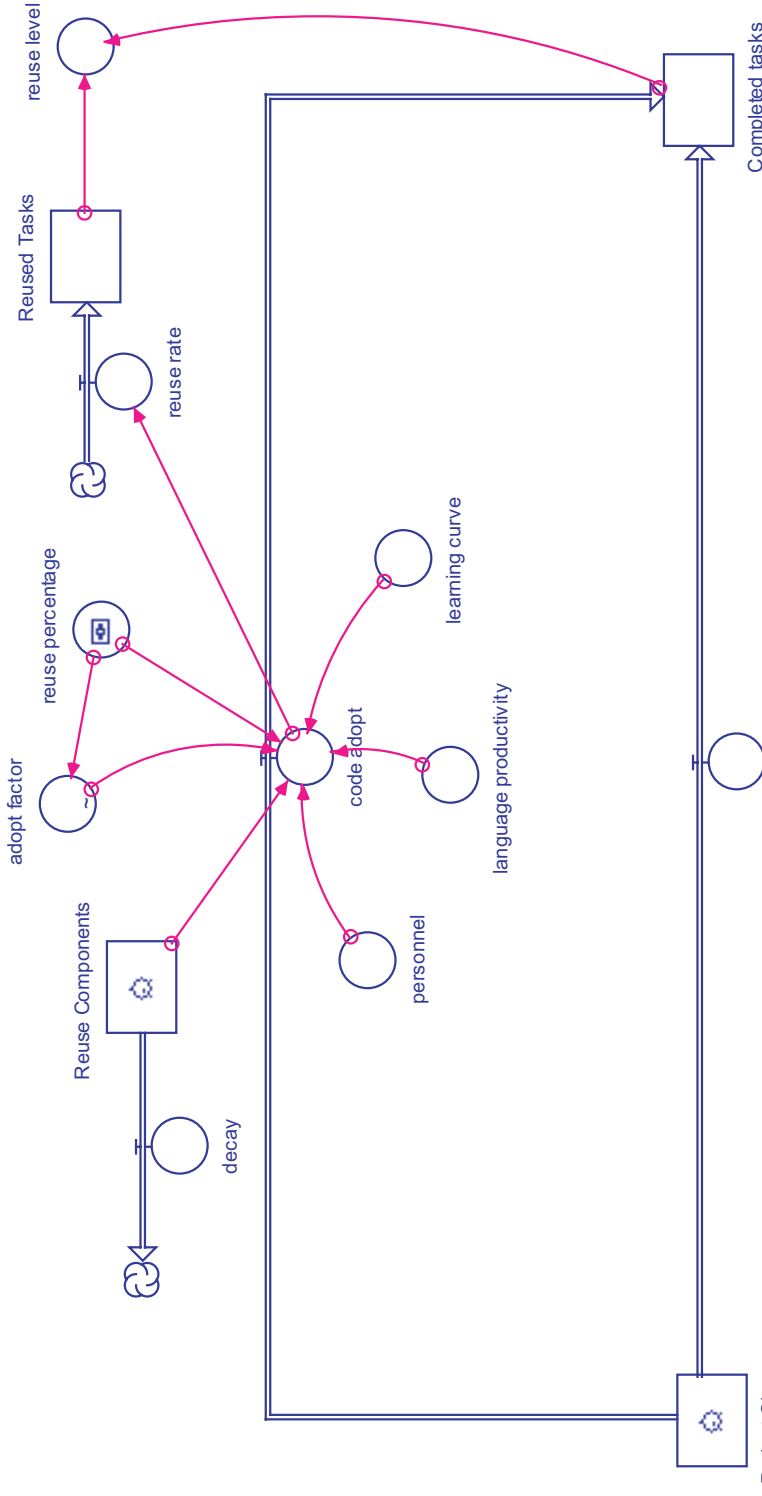


Figure 5.23. Reuse structure.

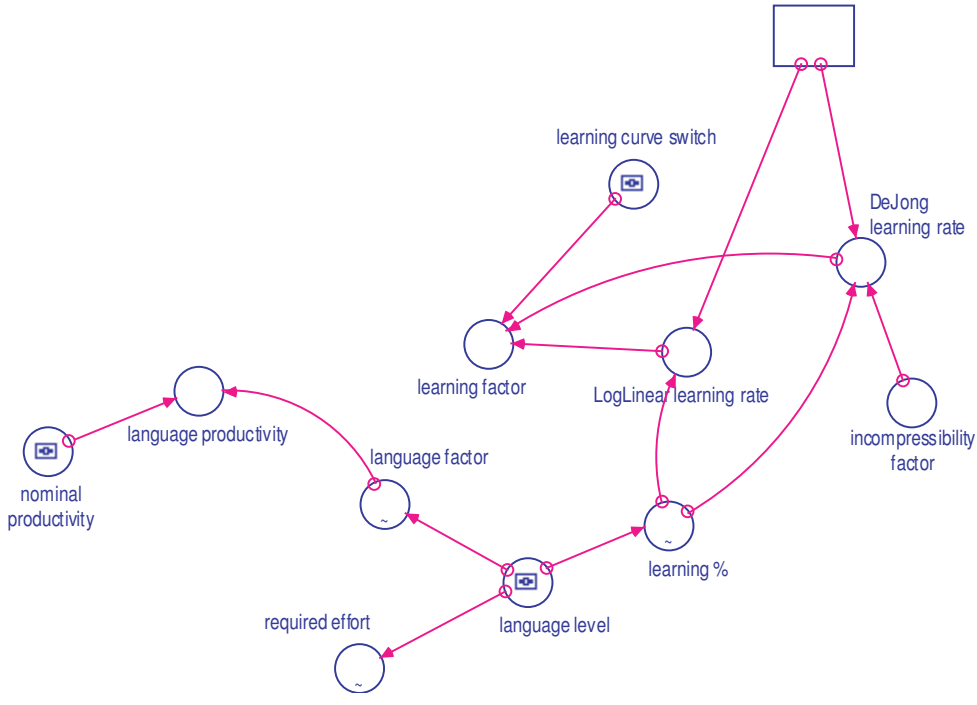


Figure 5.24. Language and learning curve.

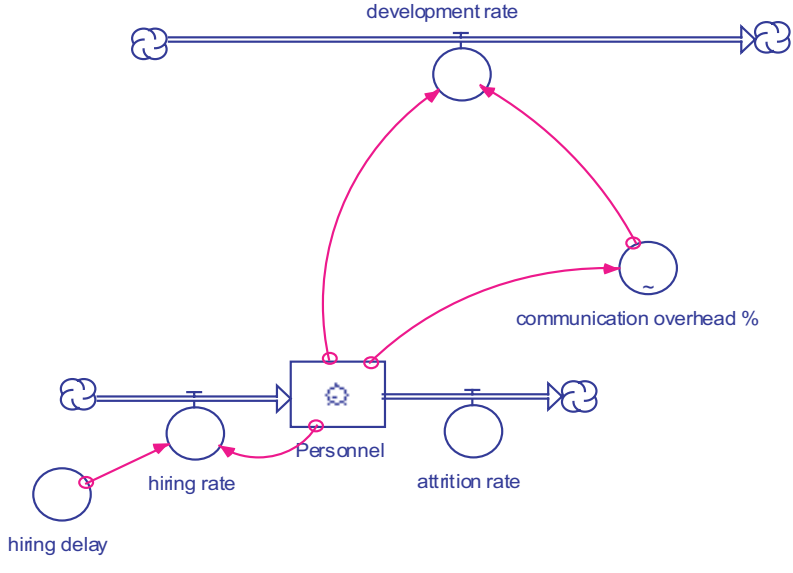


Figure 5.25. Personnel staffing structure.

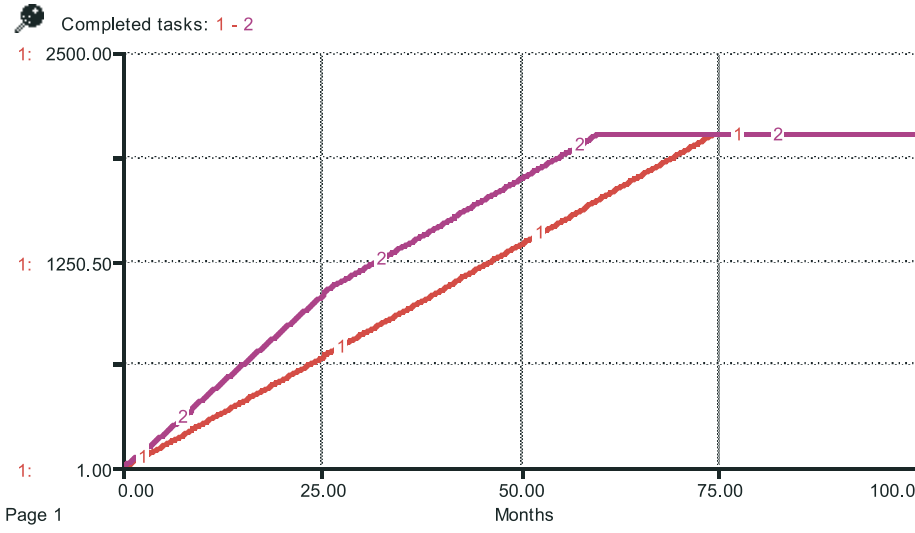


Figure 5.26. Task completion progress for no reuse versus 20% reuse (1: no reuse, 2: 20% reuse).

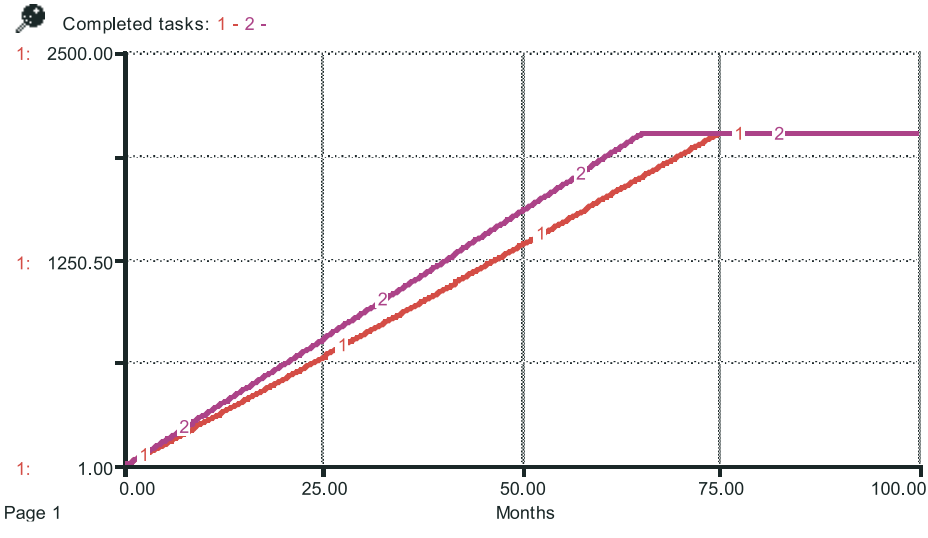


Figure 5.27. Task completion progress for 3GL versus 4GL (1: 3GL, 2: 4GL).

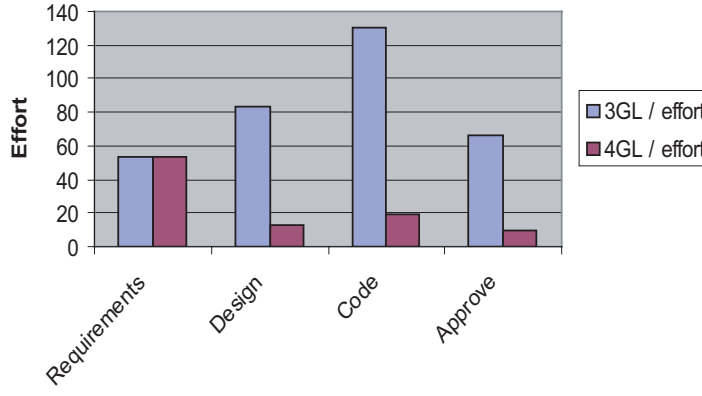


Figure 5.28. Effort distributions for 3GL versus 4GL.

Varying the number of personnel also showed tradeoffs. Optimum staff levels were found that were able to decrease overall schedule time even though communication overhead was slightly increased.

Overall, the model demonstrates that a higher language level reduces the effort on design, code, and approval phases, but it does not reduce effort in the requirements phase (Figure 5.28). With a 4GL, it also reduces the schedule time to finish the project.

5.6 COMMERCIAL OFF-THE-SHELF SOFTWARE (COTS)-BASED SYSTEMS

The use of commercial-off-the-shelf (COTS) products is becoming ever more prevalent in the creation of software systems. COTS software is commercially available as stand-alone products that offer specific functionality needed by a larger system into which they are incorporated. Such systems using COTS are frequently called COTS-based systems (CBSs). Using COTS is not just a somewhat different form of software reuse, but a fundamentally different form of software development and life-cycle management. One purpose of using COTS is to lower overall development costs and development time by taking advantage of existing, market-proven, and vendor-supported products. Shrinking budgets, accelerating rates of COTS enhancement, and expanding system requirements are all driving this process.

The definition of a COTS product per the SEI COTS-Based System Initiative* [Brownsword et al. 2000] is one that is:

*The focus of the SEI COTS-Based Systems initiative was to learn, mature, and transition principles, methods, and techniques for creating systems from COTS products. The effort has since been expanded to the creation and sustainment of systems from any set of largely off-the-shelf (rather than only commercial) constituents. The SEI has now formed the Integration of Software-Intensive Systems (ISIS) initiative to address integration and interoperability of systems of systems. The roots of ISIS are in the CBS initiative, where it became apparent that interoperability between many sorts of systems was becoming of paramount importance.

- Sold, leased, or licensed to the general public
- Offered by a vendor trying to profit from it
- Supported and evolved by the vendor, who retains the intellectual property rights
- Available in multiple identical copies
- Used without source code modification

CBSs present new and unique challenges in task interdependency constraints. USC research has shown that COTS projects have highly leveraged process concurrence between phases (see Chapter 6). Process concurrence helps illustrate how CBSs have dynamic work profiles that may quite distinct from traditional project staffing profiles.

One hypothesis about the long-term evolution dynamics of CBSs is that there are diminishing returns in trying to maximize the use of COTS components in a system development [Abts 2000]. Beyond a certain point, an increase in the number of COTS components in a system may actually reduce the system's overall economic life span rather than increase it. With simulation, one can investigate the trade-offs between maintenance costs and the increasing incompatibilities among COTS packages as they evolve to decide when to retire COTS-intensive systems. See the last example in this section on the COTS-Lifespan model for more details.

COTS glue code development presents other complex decision scenarios, and is much different from traditional glue code development. The glue code development process largely depends on factors such as the number of COTS components, volatility of the COTS components, dependability of COTS components, the interfaces provided by the COTS components, and other context factors elaborated in [Abts 2003] and [Baik et al. 2001]. Trend analyses indicate that software developers' time will be increasingly spent on glue code development and integration.

Research at USC investigated the problem of deciding the optimal time to develop glue code and integrate it into the system [Kim, Baik 1999]. This research addressing glue code development and integration processes is highlighted next.

5.6.1 Example: COTS Glue Code Development and COTS Integration

A research project by Wook Kim and Jongmoon Baik at USC sought to understand how the glue code development process and COTS integration process affect each other. They analyzed the effect on schedule and effort throughout a project life cycle of glue code development and the ongoing integration of components into the developing system. The following highlights are mostly taken from their publication [Kim, Baik 1999]. Also see [Baik et al. 2001] for additional related work. Table 5.6 is a high-level summary of the model.

5.6.1.1 Background

One generally has no control over the functionality, performance, and evolution of COTS products due to their black-box nature. Most COTS products are also not designed to interoperate with each other and most COTS vendors do not support glue

Table 5.6. COTS glue code development and COTS integration model overview

Purpose: Process Improvement, Planning
Scope: Development Project

Inputs and Parameters	Levels	Major Relationships	Outputs
• COTS Component factors • New COTS percent • Integration starting point • Glue code design productivity • Glue code development productivity • Application design productivity	• Glue code Required Designed Completed • Application software Required Designed Completed • Integrated software	• Design rates • Development rates • Integration rates • Application process concurrence • Glue code process concurrence	• Effort per activity • Schedule per activity

code (sometimes called glueware or binding code). Thus, most software development teams that use COTS components have difficulties in estimating effort and schedule for COTS glue code development and integration into application systems.

Glue code for COTS components is defined as the new code needed to get a COTS product integrated into a larger system. It is connected to the COTS component itself, acting more as a "bridge" between the COTS component and the system into which it is being integrated. It can be code needed to connect a COTS component either to higher-level system code, or to other COTS components used in the system.

Glue code is considered as one of following: (1) any code required to facilitate information or data exchange between the COTS component and the application; (2) any code needed to "hook" the COTS component into the application, even though it may not necessarily facilitate data exchange; and (3) any code needed to provide functionality that was originally intended to be provided by the COTS component, and which must interact with that COTS component.

The engineering of COTS-based systems involves significant technical risk in the glue code used to integrate components. This code is often ad hoc and brittle, but it is needed to repair mismatched assumptions that are exhibited by the components being integrated.

Most software development teams also have problems deciding when they should start to develop the glue code and integrate it into the system. It depends on factors such as the number of COTS components, requirement specification, and the availability of COTS components required for the developing system.

5.6.1.2 Model Overview

The model simulates how the integration process is affected by various factors such as the number of COTS, percentage of updated and new COTS, and requirement specification. The developing COCOTS model and data [Abts, Boehm 1998] was used for portions of

the model. COCOTS parameters calibrated for the model include COTS Product Maturity, COTS Supplier Product Extension Willingness, COTS Product Interface Complexity, COTS Supplier Product Support, COTS Supplier Provided Training, and Documentation. Data for the simulation was acquired from the COCOTS data collection program at the USC Center for Software Engineering that contained 20 projects at the time.

Staffing levels were analyzed according to concurrency profiles between glue code development and application development (or custom component development). Various starting points of glue code development were simulated to see the effect on system integration process productivity. The impact of parameters such as the ratio of new and updated COTS components, and the number of COTS components were analyzed. With these simulations, several efficient scenarios for glue code development and integration process were suggested.

Reference behavior patterns were first established to characterize the dynamic phenomena, and can be found in [Kim, Baik 1999]. In order to develop the simulation model, the following assumptions were made.

- The simulation model is focused on glue code development and its effect on the integration process, so application development and workforce effort is only briefly simulated.
- For simplicity, the simulation model does not allow feedback to the outside of the system boundary such as reevaluation of the COTS products, feedback to the requirements definition for COTS, or feedback to the COTS product selection process.

Causal loop diagrams were used to clarify the nature of the system before developing an executable model. Figure 5.29 represents a feedback diagram of the simulation model. Complete definitions of the variables in the diagram are in the original report.

Completed Glue Code increases Integrated System and Integrated System increases Glue Code Dev Multiplier. The Glue Code Dev Multipliers affect an increase in Application Design Rate. Increased Application Design Rate also increases Completed Application. Increased Completed Application also increases Integration Rate. Application Dev Rate and Glue Code Dev Rate are promoting each other.

The system dynamics model consists of four submodels: Application Development/Integration, Glue Code Development, COTS Component Factors, and Human Resource models. These are shown in Figure 5.30 with their connections. The first three submodels are highlighted below. The human resource model is not covered because it is not directly germane to the COTS phenomena, but the full details can be found in the original report [Kim, Baik 1999].

5.6.1.2.1 Glue Code Development. This sector represents glue code development in the system. COCOTS parameters for the Glue Code Development submodel are divided into categories. Personnel drivers represent characteristics of COTS integrator personnel. Application/System drivers represent characteristics of the system into which COTS is being integrated. COTS Component drivers are the most important drivers because they represent characteristics of the integrated COTS component

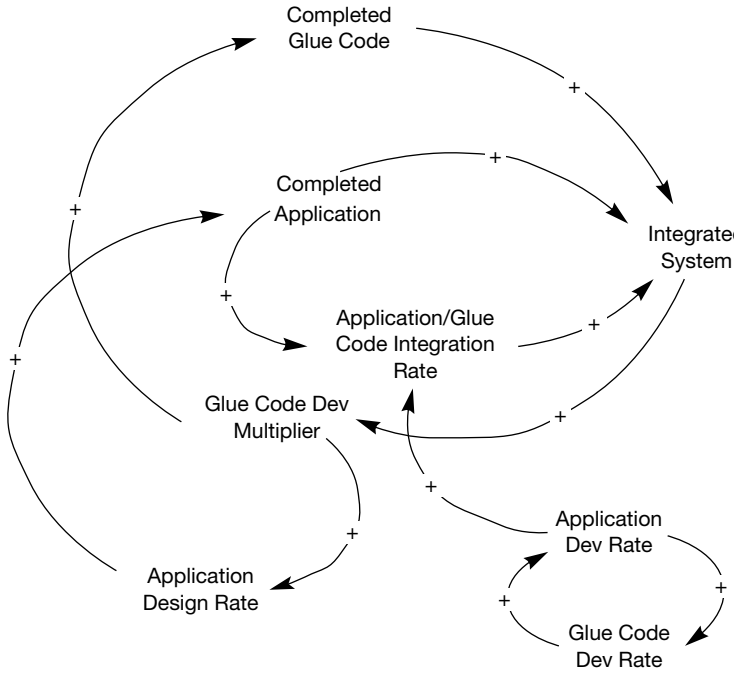


Figure 5.29. Glue code causal loop diagram.

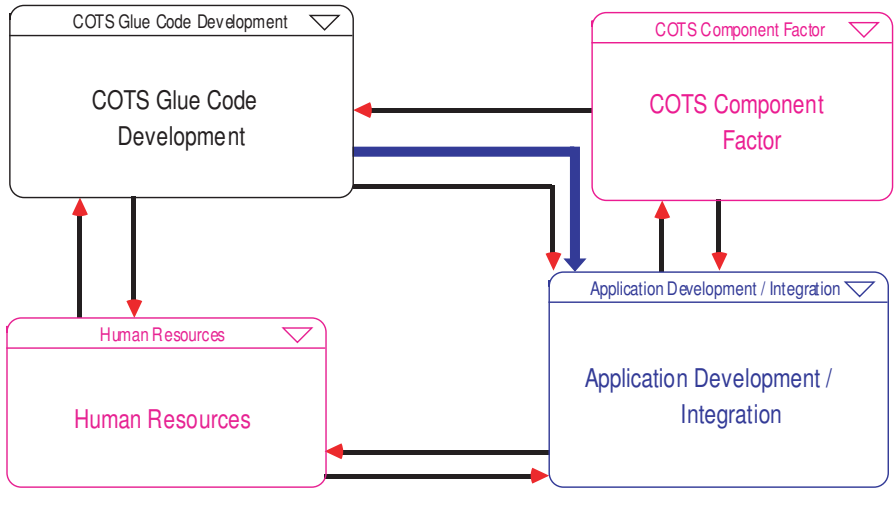


Figure 5.30. COTS glue code model sectors.

itself. These COTS Component drivers are COTS Product Maturity, COTS Supplier Product Extension Willingness, COTS Product Interface Complexity, COTS Supplier Product Support, and COTS Supplier Provided Training and Documentation.

The submodel is represented in Figure 5.31. The development process of glue code consists of three levels. The completed glue code is added to completed application code for integration. Integration process is represented in the application development submodel. In this model, COTS components are divided into new COTS components and upgraded COTS components. In the case of the upgraded COTS components, glue code is modified from a previous version. Thus, glue code development productivity is higher when integrating updated COTS components than when integrating new COTS components.

5.6.1.2.2 COTS Component Factors. This module isolates a number of factors that adjust effort in the model. They are modeled as converters with no levels or rates associated with them, so the diagram is not shown. The module contains the following COTS component factors:

1. ACPMT (COTS Product Maturity). This parameter represents COTS product maturity. The value of this parameter is estimated by time on market of the product.

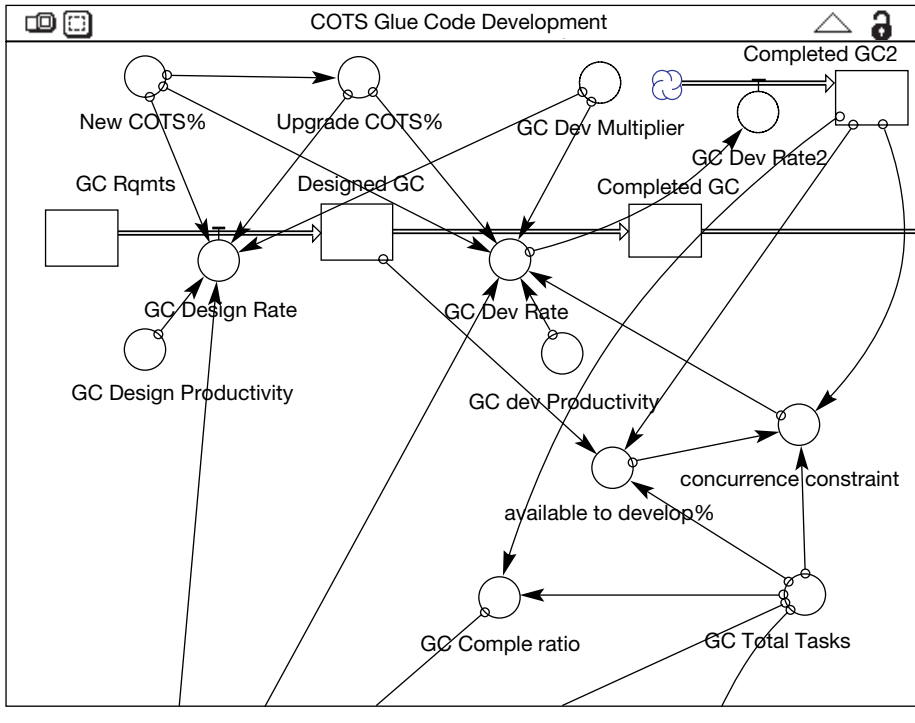


Figure 5.31. COTS glue code development model.

2. ACSEW (COTS Supplier Product Extension). COTS supplier's willingness to change features of COTS components. It is estimated by observing the number of changes the supplier makes to the COTS component and the complexity of the change.
3. APCPX (COTS Product Interface Complexity). Represents the interface complexity of a COTS product. If the interface of the COTS component is complex, it is difficult to integrate it into the application system. The degree of the complexity is calculated using a table.
4. ACPPS (COTS Supplier Product Support). COTS supplier's technical support is represented by this parameter. It includes support for the integration team during the development, either directly from the component suppliers or through third parties.
5. ACPTD (COTS Supplier Provided Training and Documentation). Training and documentation provided from a COTS supplier is enumerated in this factor. It is calculated by estimating the period of training and coverage of COTS products within documentation.

5.6.1.2.3 Application Development/Integration. The submodel in Figure 5.32 represents application development and integration in the system. The application de-

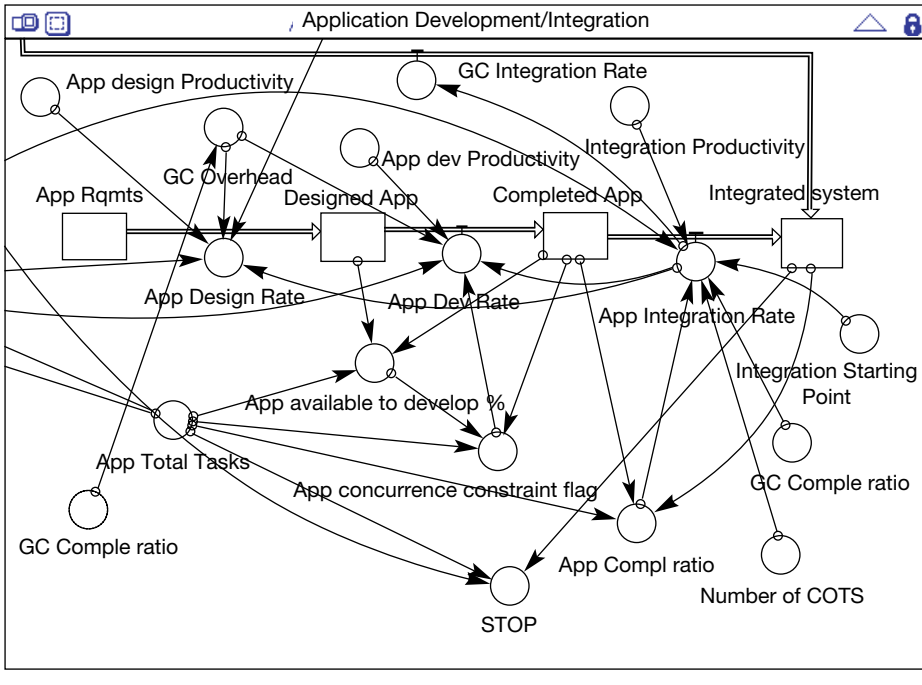


Figure 5.32. Application development/integration model.

velopment model consists of three levels. They are the same as for the glue code development. Completed application flows to the integrated system with the completed glue code. When integrating COTS components, the number of COTS components is an important factor in the integration process. If the number of COTS is higher, then the integration process is slower.

5.6.1.3 Sample Test Results

Data for these simulation results was adapted from a project in the COCOTS database. According to the graphs in Figure 5.33 and Figure 5.34, glue code development is started when parts of the application are ready to complete. There is concurrency in the design and implementation phases. The integration process starts as glue code is completed. The integration graph is S-shaped. As for personnel allocation, the staffing pattern is affected in each development phase.

According to Figure 5.33 and Figure 5.34, integration is started although glue code development is not finished. So integration can be finished just after glue code development is finished. The application development rate is reduced when the glue code development is processed because staff moves to the glue code development.

5.6.1.4 Conclusions and Future Work

One of the important results of the simulation was determining the starting point of glue code development and integration. It was found that the decision depends on factors such as the number of COTS components, requirements specification, and the availability of COTS components required for the developing system. Sensitivity analysis for a sample CBS environment showed glue code development should start

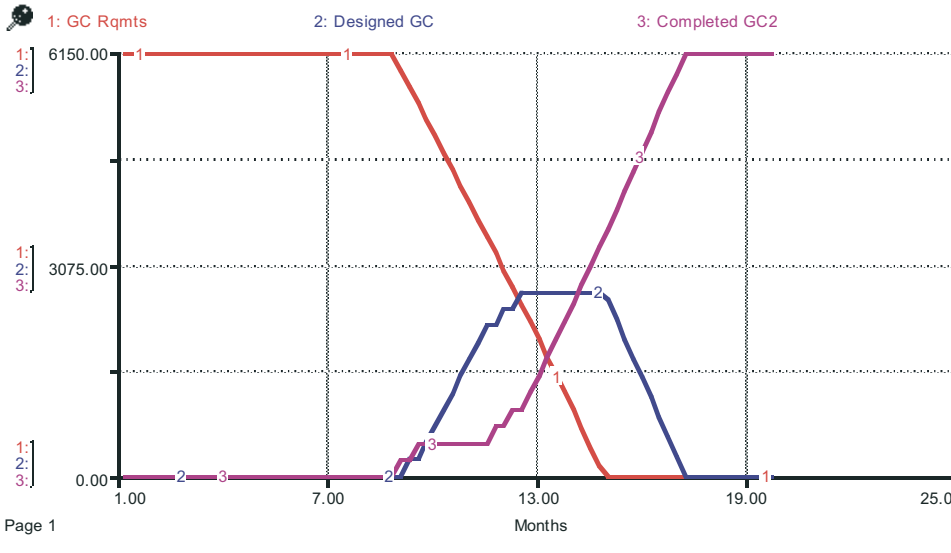


Figure 5.33. Glue code development dynamics.

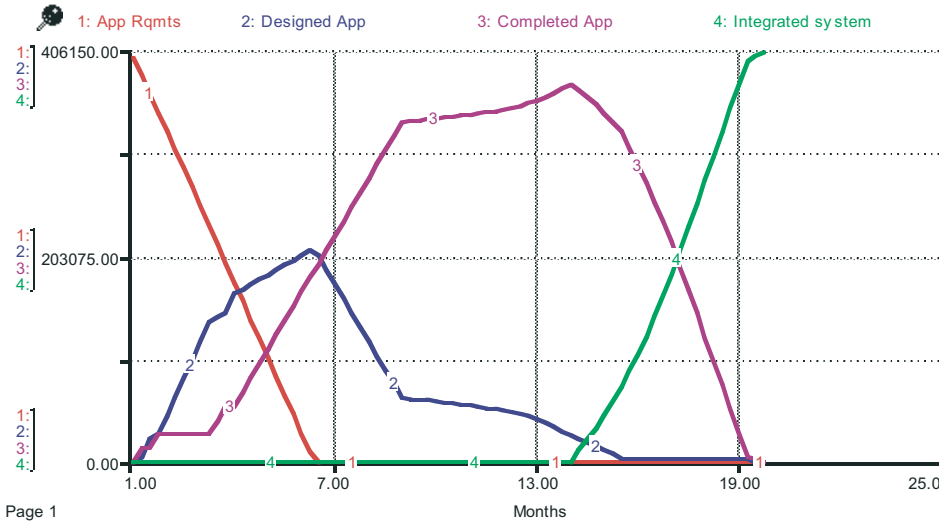


Figure 5.34. Application development and integration dynamics.

when 80% of application (custom-based) components are completed and integration should start when 30% of glue code is completed.

A general conclusion regarding the starting point is that glue code development has to start in the end of the application development, and the integration process has to start in the beginning of the glue code development.

Although software COTS products are attempting to simulate the "plug-and-play" capability of the hardware world, software COTS products seldom plug into anything easily. Most products require some amount of adaptation to work harmoniously with the other commercial or custom components in the system.

Integration of COTS software products requires adjustment and accommodations to the development process. Preparations must be made to start prototyping activities and integration activities immediately to take advantage of COTS products to accelerate development. Additional resources must be allocated late in the development cycle to provide maintenance and support to the software developers.

This simulation model can be enhanced for other aspects of the COCOTS model such as assessment, tailoring, and volatility. They include feedback to the COTS product selection process, feedback to the requirements definition for COTS products, and reevaluation of the COTS products.

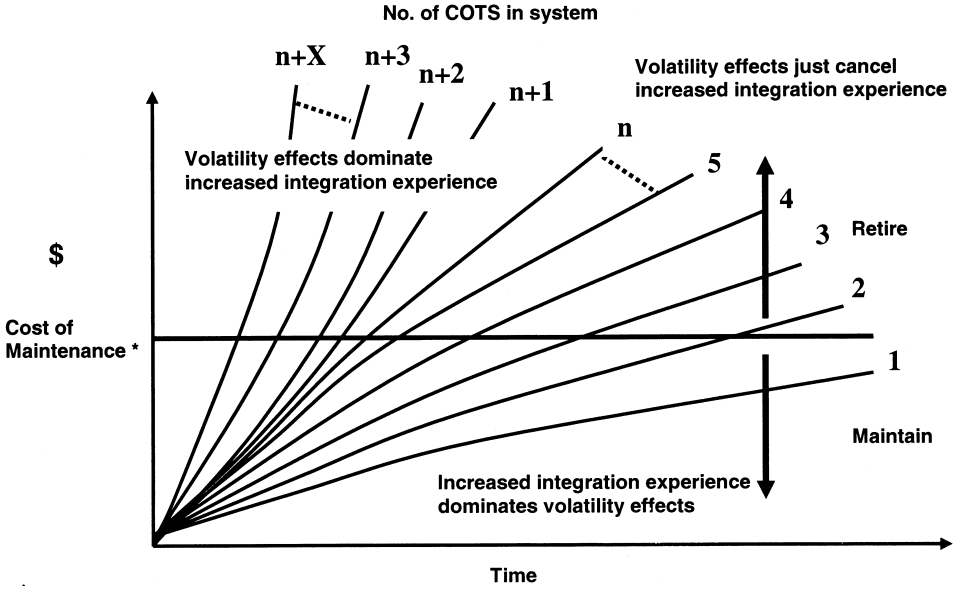
5.6.2 Example: COTS-Lifespan Model

A COTS-Lifespan Model (COTS-LIMO) was proposed in [Abts 2000]. This model is conceptual and has not yet been elaborated into an executable model. The overall concept and basic reference behaviors have been defined, such that a system dynamics model could be developed. The description below is derived from [Abts 2000].

Anecdotal evidence collected during interviews to gather calibration data for the COCOTS model suggests that generally, though not universally, the more COTS software components you include in your overall software system, the shorter the economic life will be of that system. This is particularly true when doing present-worth analyses comparing alternative designs using various combinations of COTS components, or when comparing COTS-based designs to building the entire system from scratch.

This is due to the volatility of COTS components. By volatility is meant the frequency with which vendors release new versions of their products and the significance of the changes in those new versions (i.e., minor upgrades vs. major new releases). When you first deploy your system, you have selected a suite of components that will provide the functionality you require while at the same time work in concert with each other. Over time, however, those products will likely evolve in different directions in response to the marketplace, in some cases even disappearing from the market altogether. As a consequence, the ability of these diverging products to continue functioning adequately together if and when you install the newer versions will likely also become more problematic; the more COTS components you have, the more severe the consequences of these increasing incompatibilities will become.

Looking at the model behavior itself in Figure 5.35, the graph is broken into two regions bisected by the line *n*. As long as the number of COTS components in the system is less than *n*, the increase in experience gained by your system maintainers over time and, thus, the inherent improvements in their productivity will outpace the increased effort required to maintain the system as the COTS products it contains age and evolve



*Synchronization, complexity of system, number of planned upgrades, etc.

Figure 5.35. The COTS—LIMO model behavior.

in divergent directions. However, at some number of installed COTS components *n*, the breakeven point is surpassed and no matter how skilled and experienced your maintainers become, the increases in their efficiency at maintaining the system can no longer keep pace with the impact of the increasing incompatibilities arising in the evolving COTS components. At this point, you have reached the zone in which the useful life of your system has been shortened considerably and a decision to retire the system will soon have to be made.

The actual value of *n*, the specific shape of the individual contour lines, and the location of the M–R (maintain–retire) line will be highly context sensitive, differing for each software system under review. Abts has suggested that a meta-model could be developed using systems dynamics simulation. See the exercise at the end of this chapter that addresses the lifespan trade-offs described here.

5.7 SOFTWARE ARCHITECTING

The architecture of a system forms the blueprint for building the system and consists of the most important abstractions that address global concerns. Development of an architecture is an iterative process. Architecture is important to many modern, iterative life cycles such as the Rational Unified Process (RUP) and Model-Based Architecting and Software Engineering (MBASE). It is important to develop an architecture early on that will remain stable throughout the entire life cycle and also provide future capabilities. An example process model for architecting is described next.

5.7.1 Example: Architecture Development During Inception and Elaboration

The dynamics of the architecture development process were investigated by Nikunj Mehta and Cyrus Fakharzadeh at USC. They modeled architecture development in the MBASE inception and elaboration phases for their research. The model also studies the impact of rapid application development (RAD) factors such as collaboration and prototyping on the process of architecting. The following highlights are mostly taken from [Mehta, Fakharzadeh 1999].

The primary goals of the research were to:

- Investigate the dynamics of architecture development during early MBASE life-cycle phases
- Identify nature of process concurrence in early MBASE phases
- Understand impact of collaboration and prototyping on life-cycle parameters

Table 5.7 is a high-level summary of the model.

5.7.1.1 Background

The MBASE approach involves creation of four kinds of models—product, process, property, and success models—and their integration in an ongoing fashion. MBASE is

Table 5.7. Software architecting model overview

Purpose: Process Improvement, Planning
Scope: Portion of Lifecycle—Architecting

Inputs and Parameters	Levels	Major Relationships	Outputs
• Activity effort distributions • Identified units • Average duration • Resource constraint • Defect release • Concurrence relationships	• Architecture items Identified Completed but not checked To be iterated To be coordinated Prototyping required	• Prototyping rate • Iteration rate • Basework rate • Approval rate • Initial completion rate • In-phase concurrence constraint • Interphase concurrence constraint	• Staffing profiles • Architecture items • Schedule

also an architecture-centric approach and the architecture is constantly integrated with other models.

The process of architecting is still not a very well understood one and involves varying degrees of method, theft, and intuition. However, it is possible to model the concurrence relations among requirements and architecture activities.

5.7.1.1.1 Empirical Reference Data. Substantial empirical data was available for calibrating the process model. The model behavior is predicated on data collected from CS 577 graduate student team projects at USC. The data for the model was acquired from the CSCI 577a (Fall 1998) and CSCI 577b (Spring 1999) Weekly Effort Reports. The data is for effort for a standard set of activities, size (documentation), use cases, and requirements.

A summary of some of the data is shown in Table 5.8. It shows the individual activity person-hours and percentages to reach LCO, LCA, a Revised LCA (RLCA), LCA

Table 5.8. CSCI 577 effort distribution data

	LCO	%	LCA	%	RLCA	%	Total LCA	%	Overall	%	Adjusted
Management	230.8	40%	235	41%	112	19%	348	60%	578	38%	
Environment	110.1	60%	18	10%	55	30%	72	40%	182	12%	
Requirements	173.9	53%	103	32%	48	15%	152	47%	325	22%	50%
Architecture and Design	101	45%	64	28%	61	27%	125	55%	226	15%	34%
Implementation	36	36%	30	30%	33	34%	63	64%	99	7%	16%
Assessment	35	35%	41	40%	25	25%	66	65%	100	7%	
Total	686	45%	490.8	32%	334.7	22%	826	55%	1511.6	100%	100%

total, and overall. The adjusted column normalizes the percentages for the activities represented in the model. The entire dataset had more detailed activities. The original report has the full dataset, defines the activities in detail, and describes the effort adjustment algorithms.

5.7.1.2 Model Overview

The model simulates the two major activities of the front end of a software development project: requirements elicitation and architecture design. It also models the prototyping tasks in the development process as a supporting activity. Some features of the model include:

- Schedule as independent variable
- Iterative process structures
- Sequentiality and concurrency among phases (requirements and architecture/design) and activities (initial completion, coordination, quality assurance, iteration)
- Demand-based resource allocation
- External and internal precedence constraints

These features are addressed in the following sections.

The two activities are modeled separately in the model and a generic process structure is chosen to describe the internal dynamics of each activity with customizations performed to accommodate each activity's characteristics. The resource allocation in a concurrent model cannot be described by a static relationship with progress. Instead, the required resources are often dictated by resource availability. Once hired, these resources are dynamically allocated to various tasks based on the backlog of tasks of that kind. Three subsystems of this model are process structure, resources, and performance.

The flow of products between project activities is described in the project network diagram in Figure 5.36. Links between activities of the project are distinguished as car-

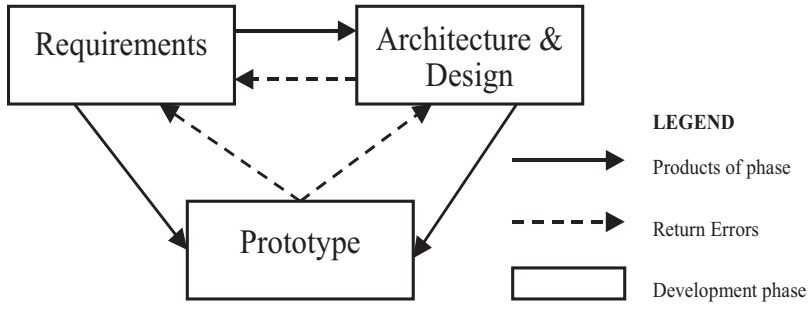


Figure 5.36. Activity network diagram.

rying products of an activity or returning errors from the following activity. The inter-activity interactions arise from the following:

- Requirements activity produces requirement descriptions that are available to the architecture activity.
- Architecture activity produces artifacts that are used in downstream activities but these downstream activities are not modeled.
- Architecture design reveals errors in the requirements definition that cause rework in the previous phase.
- Prototyping activity is driven by both requirements and architecture activities and serves as a fast-track route for discovering the details that would otherwise require more effort and time.
- Once prototyping is performed, it uncovers information that is required to better describe and understand the products of each of the two principal activities. Prototyping is a supporting activity and does not by itself consume or produce any artifacts (assumed in this model, but prototypes are not always throwaways).

The model is based on the assumptions that the projects are completed in a fixed time frame and that schedule is an independent variable. This assumption is necessary so that the model can be calibrated against a set of projects from the CSCI 577 courses.

Modeling a concurrent process requires a good understanding the relations of the various concurrent activities of the project. The model is based on the product development project model by Ford and Sterman [Ford, Sterman 1997], which describes the concurrency relationships that constrain the sequencing of tasks and interactions with resources. This model identifies a generic process structure that can be used to describe the iteration and completion of artifacts and the generation and correction of errors in each activity.

Process elements are organized in the form of a phase-activity matrix, so that both requirements elicitation and architecture design activities involve initial completion, coordination, QA, and iteration. All the rates and levels of the generic process structure are arrayed in two dimensions: activity and tasks.

The concurrence for requirements elicitation indicates that it experiences a fairly high rate of independence on its state of completion. The architecture concurrence is in the form of an S-shape curve. This indicates that there is a significant dependence of architecture on the requirements.

The only interactivity dependency is that between the requirements and architecture activity. It is modeled as shown in Figure 5.37, showing the fraction of architecture tasks available to complete the given completed requirements. The S-shaped curve starting at the origin indicates that the architecture design can start only after some requirements elicitation is performed. However, the steep incline of the curve indicates that once some information is available about the requirements, the rest of the architecture can be designed quickly.

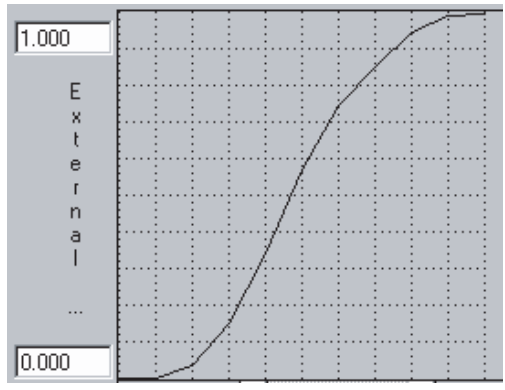


Figure 5.37. External concurrence constraint for architecture design activity (fraction of architecture tasks available to complete versus requirements completed).

The rate of iteration is determined by both the resource constraints on this rate as well as the minimum activity duration for iteration. A separate prototyping chain is created to handle the demand for prototyping and create the prototypes based on resource and process constraints.

The feedback loop diagram for the model is shown in the Figure 5.38. It shows the reinforcement feedback loop caused by initial basework when more items are completed as some items get completed. Another reinforcement loop is created due to prototyping; as more initial work is completed, the need for prototyping increases and, thus, the iteration rate increases as a result.

The main structures in the model are seen in Figure 5.39. Plots from some test runs are shown in Figure 5.40 and Figure 5.41. The first graph shows the levels of architecture items in different stages, whereas the second one shows the necessary personnel resources over time needed to support the architecting process.

5.7.1.3 Conclusions and Future Work

The model shows that initial completion rates for the requirements identification and architecture development activities significantly impact the number of approved items. Prototyping factors such as IKIWISI and collaboration also significantly affect the rates of completion and approval. The model also describes a declining curve for staffing analysts and a linear growth for architecting and design personnel. The model behavior is similar to RUP in terms of the effort distribution curves of the process.

This model is able to replicate the effort profiles for requirements and architecture/design activities based on a concurrent development model and a dynamic resource allocation scheme. Our model also provides a starting point to model many RAD opportunity factors for understanding the effects of RAD techniques on software life cycles. In essence, this model can be a handy test bed for designing

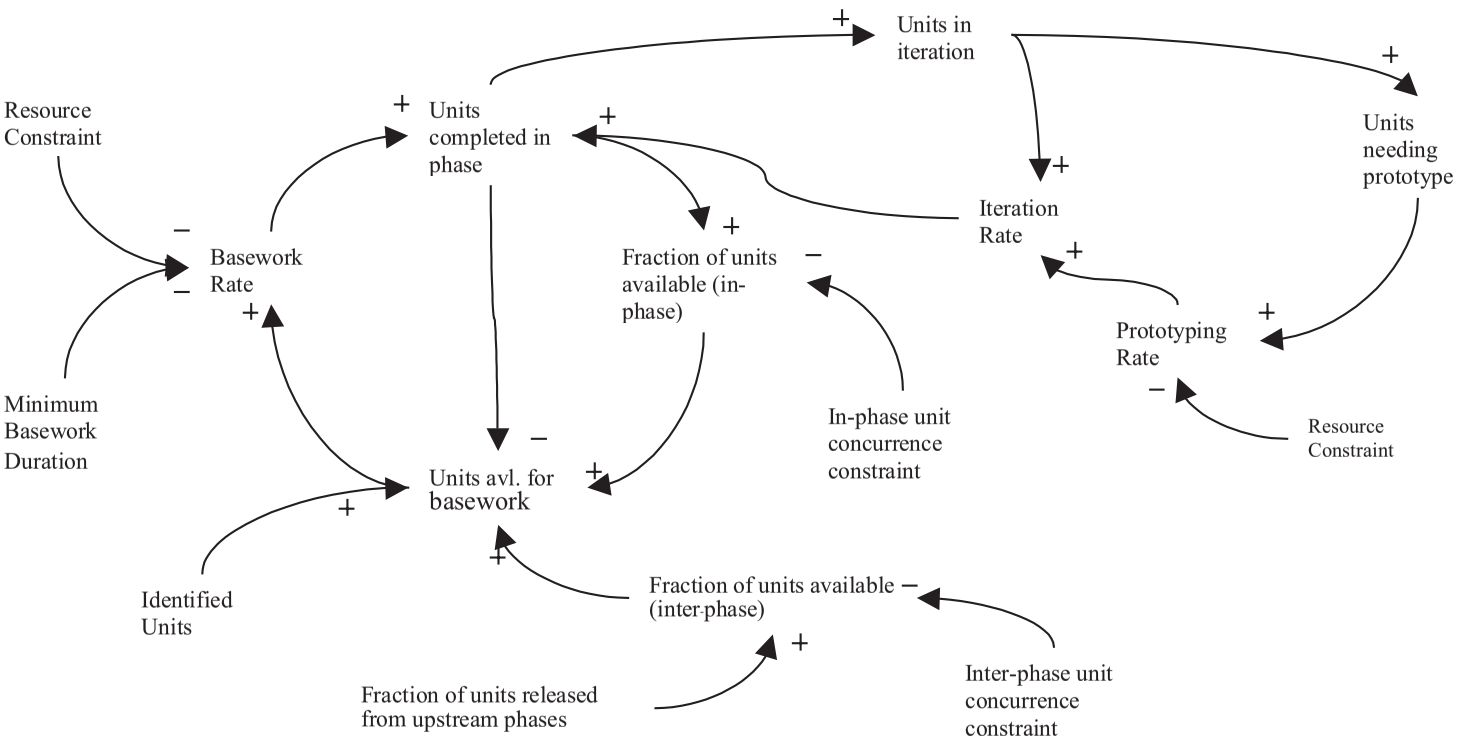


Figure 5.38. Main causal loop.

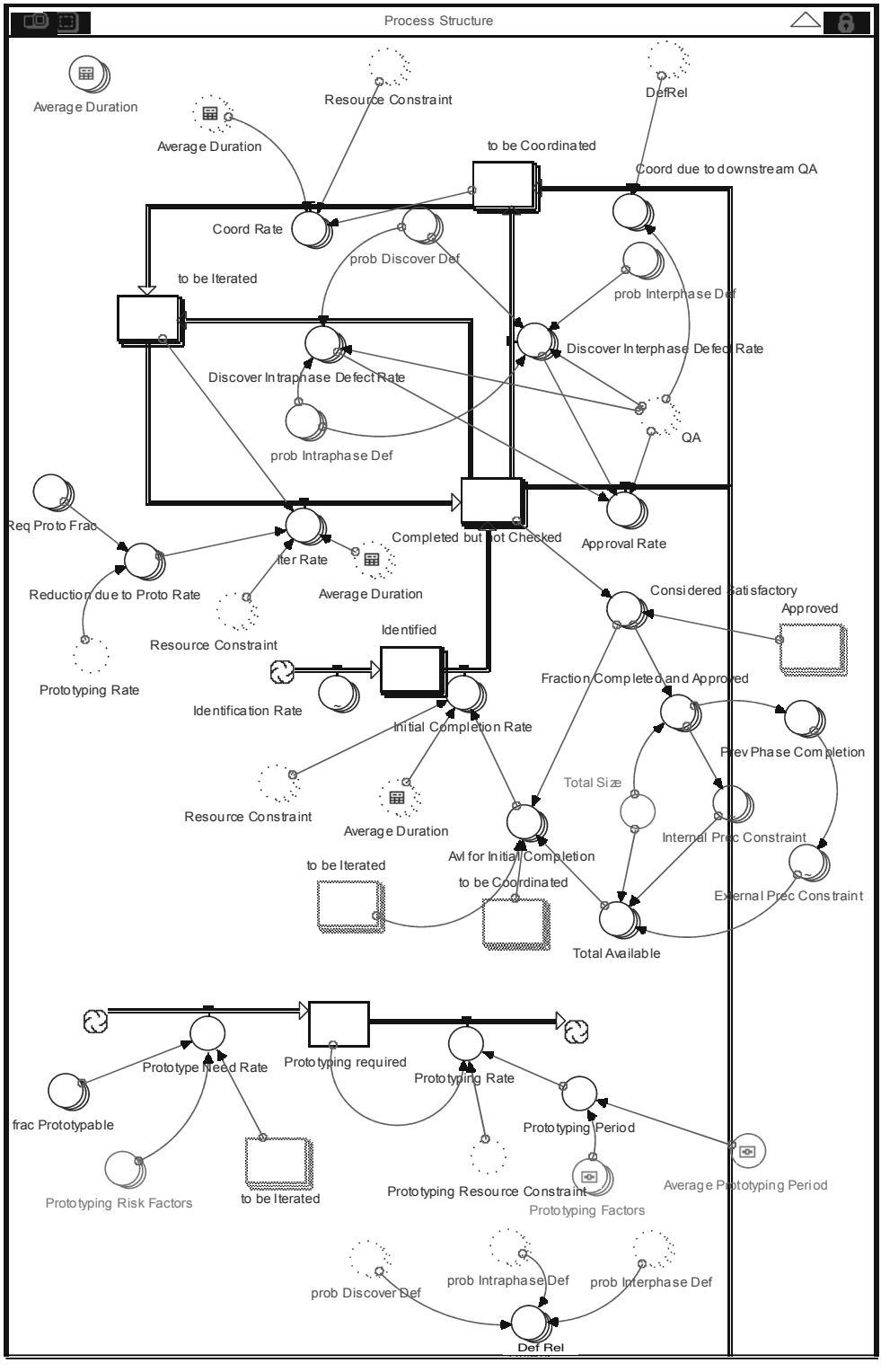


Figure 5.39. MBASE architecting model.

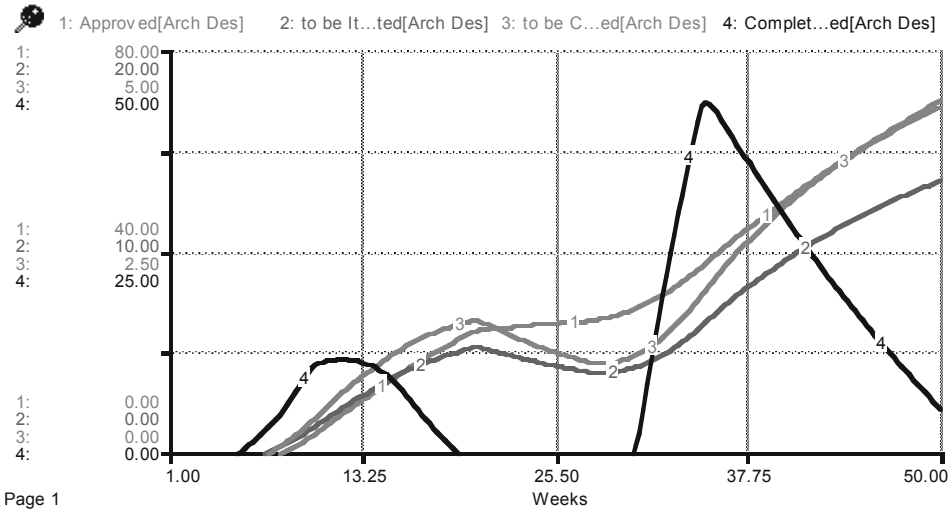


Figure 5.40. Architecture items.

an RAD process model. It can also be used to study staffing patterns for RAD projects.

As future work, it would be possible to study the creation and removal of defects in the model. Currently these are statically chosen and no probabilistic approach has been taken. The effect of peer reviews and walk-throughs on the defect rates can also serve as a major addition to the model.

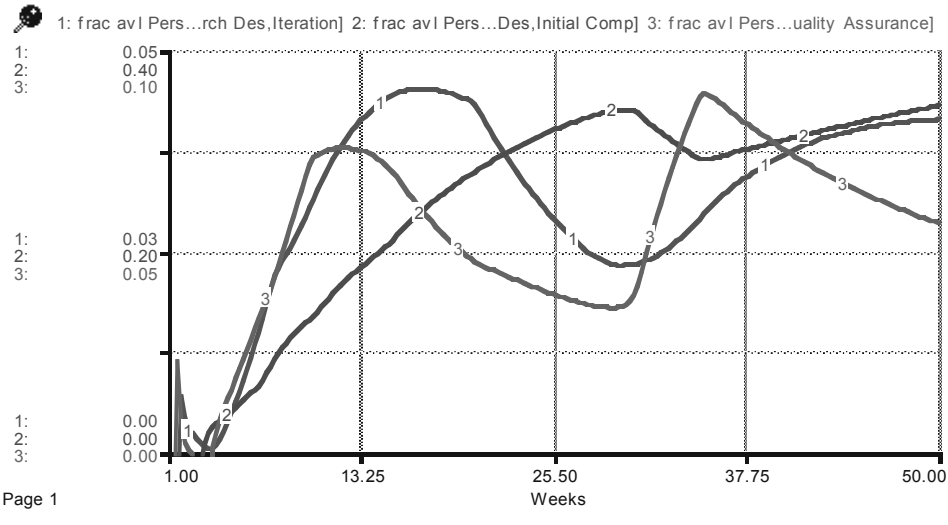


Figure 5.41. Architecture personnel fractions.

5.8 QUALITY AND DEFECTS

Software quality means different things to different people. The relative importance of different characteristics or attributes depends on who is assessing the software and what they need or expect from it. Utility can be considered in many ways. Users frequently are concerned about the number and type of failures that occur during usage. Hence, the number of defects is a measure of quality but there are many other possible measures.

A list of representative quality attributes was introduced earlier in the chapter in Section 5.1. Working definitions of these attributes are provided in [Madachy, Boehm 2005] using a value-based stakeholder perspective. Product attributes are not always independent and the relationship between technologies for achieving product measures is not easy to model. See [Boehm et al. 2004] for further details and discussion of the attributes.

The value-based product model described in detail in Chapter 6 addresses the attributes *reliability, affordability,* and *timeliness* in a commercial market context. The stakeholder business value being optimized is profit, and the goal is to maximize profit from investing in processes that contribute to reliability. The model is primarily covered in the organizational applications chapter because the primary result variables used to gauge the decision options are organizational financial measures.

However, there is a major relationship between product characteristics and their impact on business value. The quality of a product is a primary factor in sales. Achieving revenue from a reliable product is balanced against its affordability and timeliness of delivery. The model demonstrates a value-based framework for decision analysis by modeling dependability impact on costs and profit of achieving different reliability levels.

In order to assess quality, there should be provisions for representing defects since latent ones may impact a system with respect to quality attributes. A way to model different defect severities or types of defects using system dynamics is to have separate flow chains to represent the different categories.

The number of defects is generally considered a rough measure of overall quality, but is most closely tied to the attribute *correctness* (depending on the stakeholder perspective). Modeling the resources expended on defect detection and removal supports trade-off decisions to achieve product goals.

Defect analysis is a primary strategy for facilitating process improvement. Defect categorization helps identify where work must be done and to predict future defects, whereas causal analysis seeks to prevent problems from reoccurring. Defect prevention is a highly relevant area that modeling and simulation can specifically address. It is a high-maturity, key process area in process improvement frameworks including the SW CMM and CMMI.

A number of organizations are using process modeling as a preventive measure to improve their process performance. One example is Motorola, which has been modeling defect generation and prevention as part of their overall efforts to improve software processes [Eickelmann et al. 2002]. They have used combined approaches that use discrete aspects to tie a variety of attributes to defects, including different severity levels

and corresponding defect-finding effectiveness values. They have also used ODC attributes in their process modeling. The advantage of assigning detailed attributes is a primary reason that companies addressing defect prevention resort to discrete event or combined modeling.

5.8.1 Example: Defect Dynamics

The integrated model developed by Abdel-Hamid [Abdel-Hamid, Madnick 1991] included defect flows and interventions, including quality assurance and testing. Table 5.9 is a high-level summary of the QA sector model.

The flow chains in Figure 5.42 were used to model the generation, detection, and correction of errors during development. The chains are simplified and only show the connections to adjacent model elements. There are two types of errors in the model, called passive and active. Active errors can multiply into more errors whereas passive ones do not. All design errors are considered active since they could result in coding errors, erroneous documentation, test plans, and so on. Coding errors may be either active or passive.

There is a positive feedback loop between the undetected active errors and the active error regeneration rate. Potentially detectable errors are fed by an error generation rate. The errors committed per task is defined as a function against the percent of job worked. The workforce mix and schedule pressure also affect the error generation rates. The model addresses both the growth of undetected errors as escaped errors and bad fixes that generate more errors, and the detection/correction of those errors. Figure 5.43 shows a graph of some of the important quantities for a representative simulation run.

Table 5.9. Quality assurance sector model overview

Purpose: Planning, Process Improvement
Scope: Development Project

Inputs and Parameters	Levels	Major Relationships	Outputs
• Daily manpower for QA (from manpower allocation sector) • Daily manpower for rework (from manpower allocation sector) • Multiplier for losses (from software productivity sector) • Percent of job worked (from control sector) • Average QA delay	• Tasks worked • Errors Potentially detectable Detected Reworked Escaped	• Error generation rate • Nominal errors committed • Error multiplier due to schedule pressure • Error multiplier due to workforce mix • Nominal QA manpower needed per error • Nominal rework manpower needed per error • Multiplier due to error density	• Detected errors

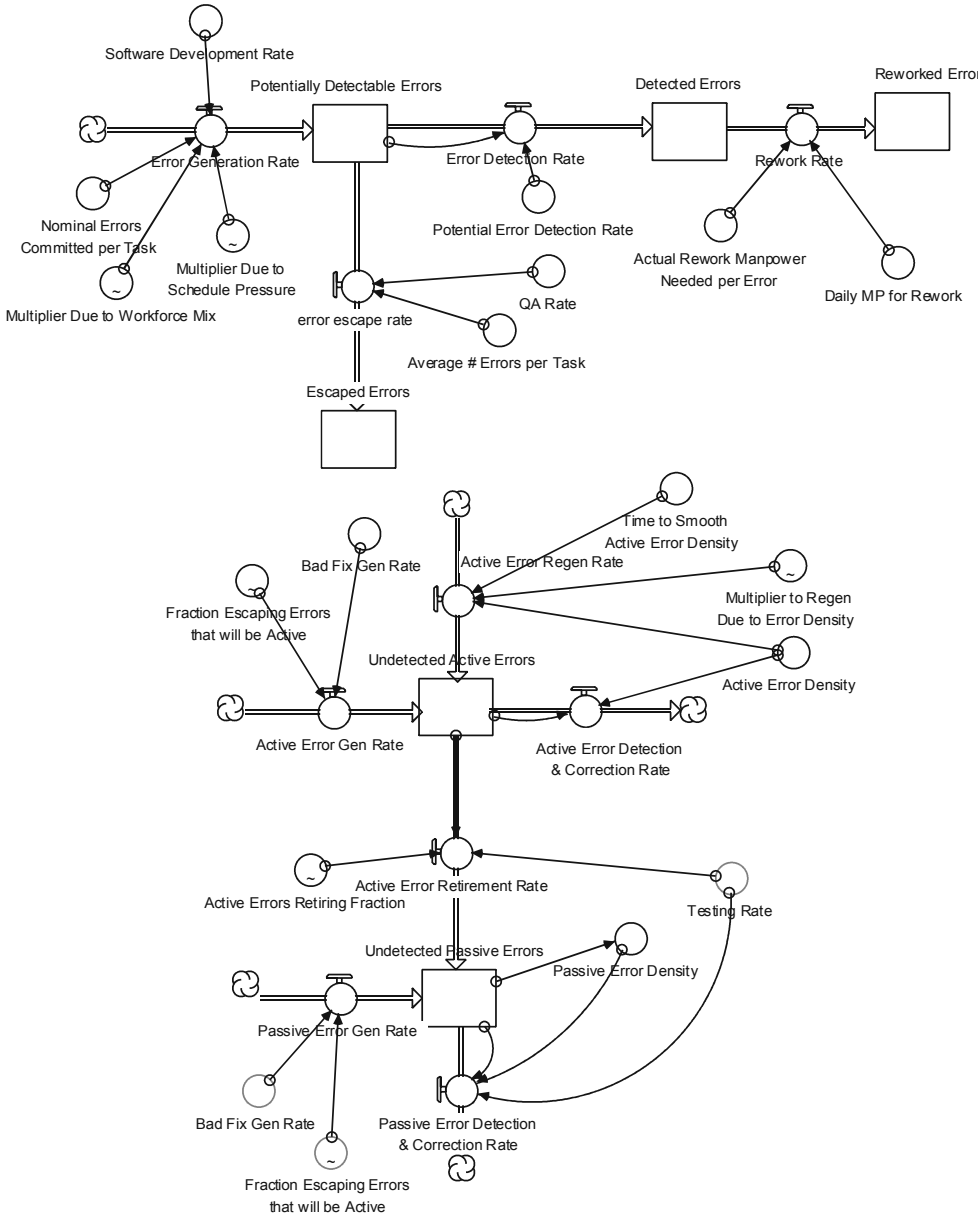


Figure 5.42. Integrated project model defect flow chains.

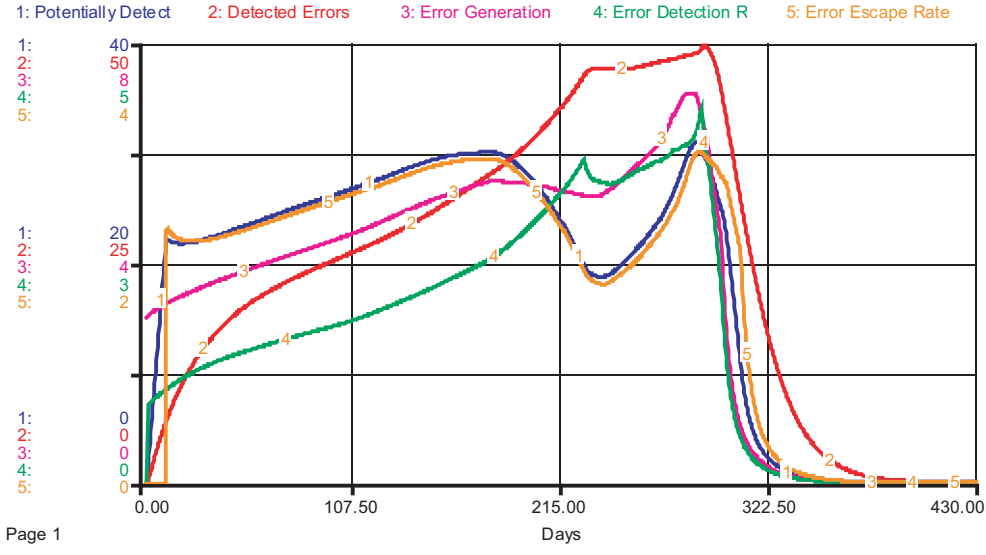


Figure 5.43. Integrated project model defect dynamics (1: Potentially Detectable Errors, 2: Detected Errors, 3: Error Generation Rate, 4: Error Detection Rate, 5: Error Escape Rate).

The error detection rate is a function of how much effort is spent on QA. It is assumed that easy, obvious errors are detected first, and that subsequent errors are more elusive and expensive to find.

System testing is assumed to find all errors escaped from the QA process and bad fixes resulting from faulty rework. Any remaining errors could be found in maintenance, but is not included in the model.

A representative example of determining "how much is enough" was provided by the model. The optimal amount of quality assurance activities was experimentally determined to address the attributes of *affordability* and *timeliness.* The trade-offs are shown in Figure 5.44.

Too much quality assurance can be wasteful, yet not enough will impact the effort and schedule adversely because defects will get through. The key is to run a simulation at applicable values across the spectrum and determine the optimum strategy or sweet spot of the process. Based on their assumptions of a project environment, about 15% of the total project effort was the optimal amount to dedicate to quality assurance.

5.8.2 Example: Defect Removal Techniques and Orthogonal Defect Classification

At USC, we are currently developing simulation models for NASA* to evaluate the effectiveness of different defect detection techniques against ODC defect categories.

*This work is sponsored by NASA Ames Cooperative Agreement No. NNA06CB29A for the project *Software Risk Advisory Tools.*

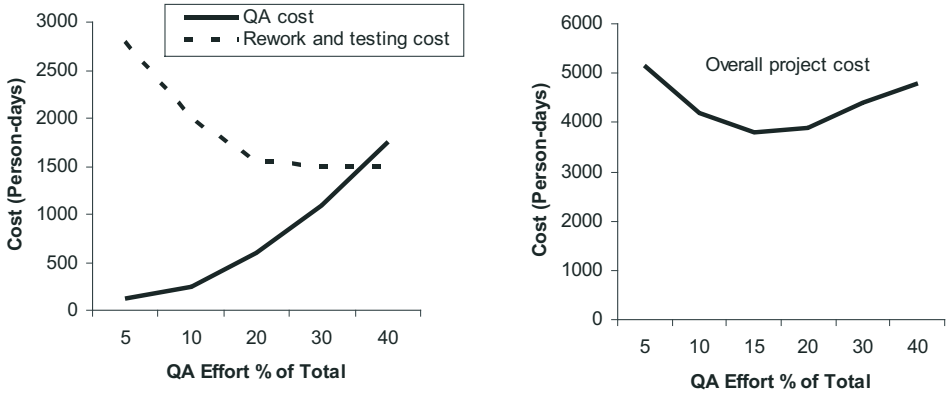


Figure 5.44. Quality assurance trade-offs.

This section summarizes a continuous-risk model that extends the Constructive Quality Model (COQUALMO) [Chulani, Boehm 1999]. The model uses standard COCOMO factors for defect generation rates and defect removal techniques for automated analysis, peer reviews, and execution testing and tools. A summary of the simulation model is in Table 5.10.

COQUALMO is a static model, which is a form not amenable to continuous updating because the parameters are constant over time. Its outputs are final cumulative quantities, no time trends are available, and there is no provision to handle the overlapping capabilities of defect detection techniques. The defect detection methods (also called V&V techniques) are modeled in aggregate and it is not possible to deduce how many are captured by which technique (except in the degenerate case, where two of the three methods are zeroed out).

The primary extensions to COQUALMO include:

- Defect and generation rates are explicitly modeled over time with feedback relationships.
- The top-level defects for requirements, design and code, are decomposed into ODC categories.
- The model is calibrated with relevant NASA data.
- There are trade-offs of different detection efficiencies for the removal practices per type of defect.

The system dynamics version can provide continual updates of risk estimates based on project and code metrics. The simulation model is easily updated as more data becomes available and incorporated into the calibration. The current ODC defect distribution pattern is per JPL studies [Lutz, Mikulski 2003]. This model includes the effects of all defect detection efficiencies for the defect reduction techniques against each ODC defect type.

Table 5.10. Model summary table

Purpose: Process Improvement, Planning, Control, and Operational Management
Scope: Development Project

Inputs and Parameters	Levels	Major Relationships	Outputs
• SLOC • Effort coefficient • Schedule coefficient • Automated analysis setting • Peer review setting • Execution testing and tools setting • 180 defect detection efficiencies (10 defect types – 3 defect reduction techniques · 6 settings per technique) • Generation buildup parameter (3) • Detection buildup parameter (3) • Nominal design start time • Nominal code start time	• Defects Requirements—Correctness Requirements—Completeness Requirements—Consistency Requirements—Ambiguity/Testability Design/Code—Timing Design/Code—Class/Object/Function Design/Code—Method/Logic/Algorithm Design/Code—Data Design/Code—Values/Initialization Design/Code—Checking • Defects found Requirements—Correctness Requirements—Completeness Requirements—Consistency Requirements—Ambiguity/Testability Design/Code—Interface Design/Code—Timing Design/Code—Class/Object/Function Design/Code—Method/Logic/Algorithm Design/Code—Data Design/Code—Values/Initialization Design/Code—Checking	• Defect generation rates (10 defect types) • Defect detection rates (10 defect types) • Defect elaboration functions • Composite defect detection efficiencies	• Defect generation curves (10 defect types) • Defect detection curves (10 defect types) • Final defects (10 defect types) • Effort • Schedule

Figure 5.45. Defect removal settings on control panel (1 = very low, 2 = low, 3 = nominal, 4 = high, 5 = very high, 6 = extra high.

The defect removal factors are shown in the control panel portion in Figure 5.45. They can be used interactively during a run. A simplified portion of the system diagram (for completeness defects only) is in Figure 5.46. The defect dynamics are based on a Rayleigh curve defect model of generation and detection. The buildup parameters for each type of defect are calibrated for the estimated project schedule time, which may vary based on changing conditions during the project.

The defect detection efficiencies are modeled for each pairing of defect removal technique and ODC defect type. Examples of how the defect detection techniques have different efficiencies for different defect types are shown in Figure 5.47 and Figure 5.48. Peer reviews, for instance, are good at finding completeness defects in requirements but not efficient at finding timing errors for a real-time system. Those are best found with automated analysis or execution and testing tools. These differences for peer reviews are reflected in the graphs for defect detection efficiency for the different defect types.

The scenario is demonstrated in Figure 5.49 through Figure 5.51, showing the dynamic responses to changing defect removal settings on different defect types. Figure 5.49 shows the defect removal settings with all defect removal practices initially set to nominal. At about 6 months, automated analysis goes high and then is relaxed as peer reviews is kicked up simultaneously. The variable impact of the different defect types can be visualized in the curves.

The requirements consistency defects are in Figure 5.50, showing the perturbation from the defect removal changes. The code timing defects is evident in Figure 5.51. Near the end, the setting for execution testing and tools goes high, and the timing defect detection curve responds to find more defects at a faster rate.

5.9 REQUIREMENTS VOLATILITY*

Requirements volatility refers to growth or changes in requirements during a project's software development life cycle. It could be due to factors such as mission or user interface evolution, technology upgrades, or COTS volatility. The requirements volatility phenomenon is a common ailment for an extremely high percentage of software pro-

*Ashwin Bhatnagar initially created this section on requirements volatility as a graduate student at USC and wrote most of the description of the software project management simulator example.

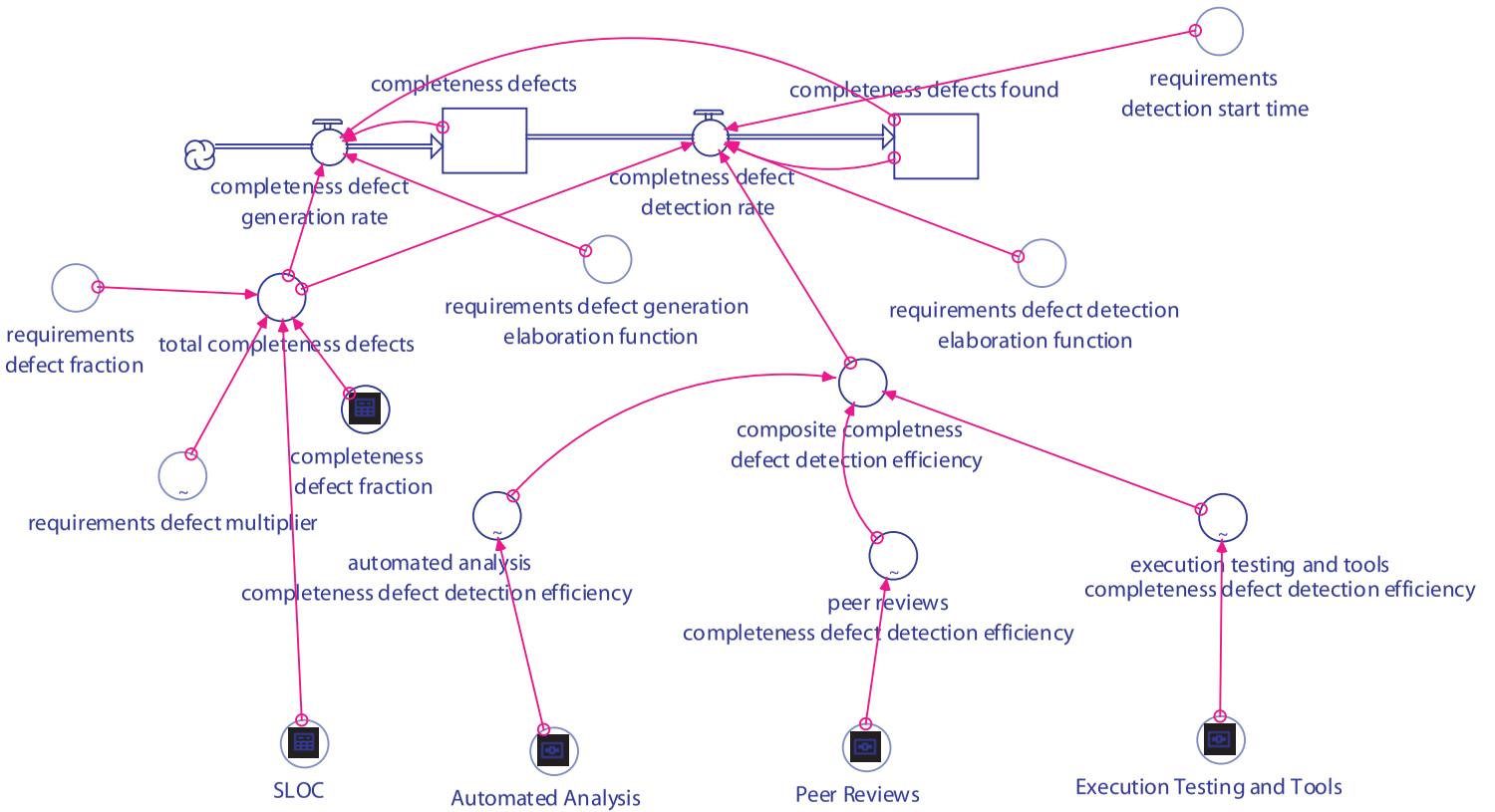


Figure 5.46. Model diagram portion (completeness defects only).

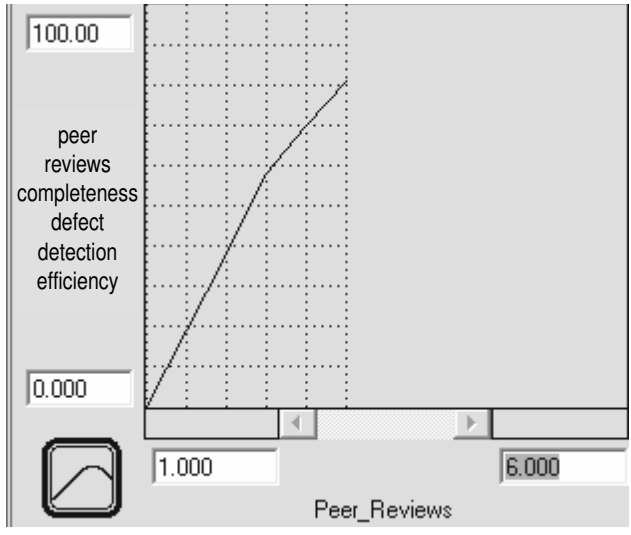


Figure 5.47. Peer review detection efficiency for requirements completeness defects.

jects. In addition, the effects of requirements volatility on critical project success factors such as cost, schedule, and quality are not well documented nor well understood.

The situation is made more complicated by the growth of the IKIWISI (I'll Know It When I See It) syndrome. Traditional waterfall life-cycle philosophy dictated that requirements had to be frozen before the design of the project was initiated, but this form

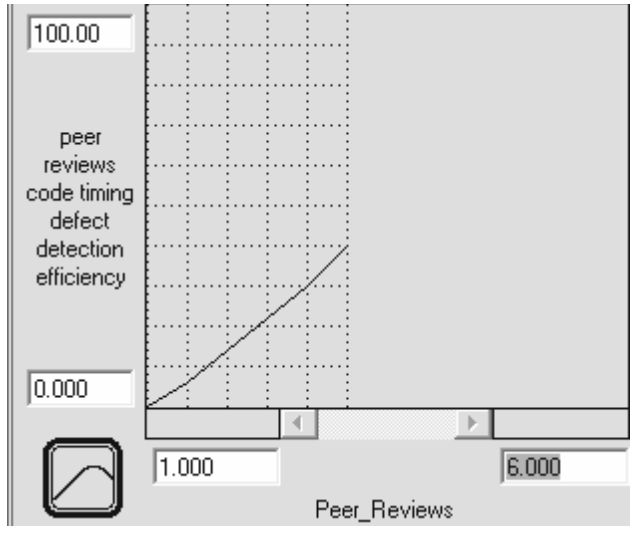


Figure 5.48. Peer review detection efficiency for code timing defects.

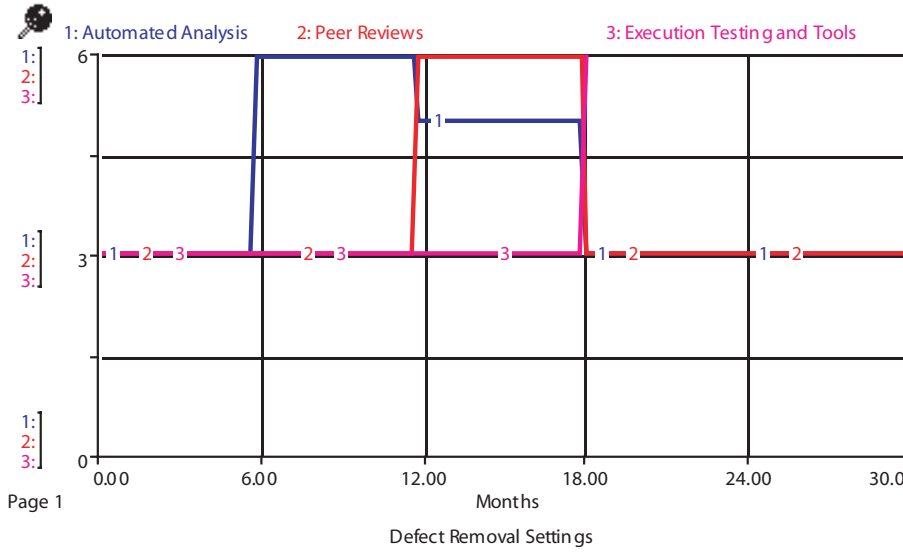


Figure 5.49. Defect removal settings.

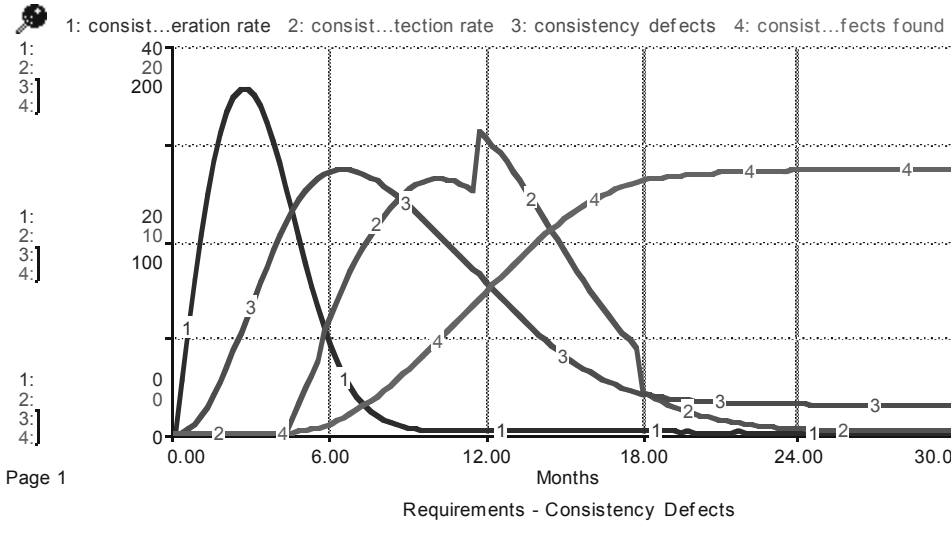


Figure 5.50. Requirements—consistency defects.

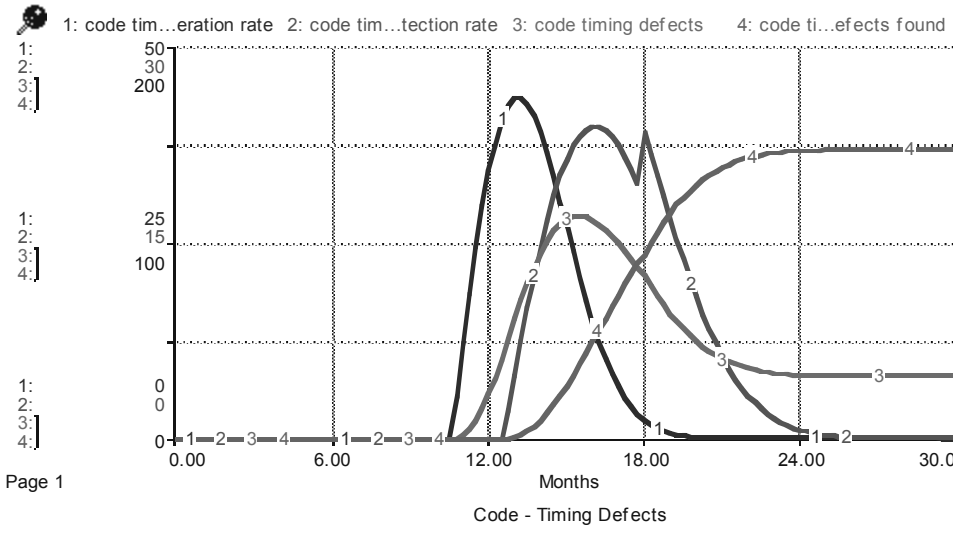


Figure 5.51. Code—timing defects.

of project development is unrealistic [Reifer 2000]. Requirement changes can be initiated in any phase of the software development life cycle and are unavoidable [Kotonya, Sommerville 1998]. Requirement changes can even be introduced after project completion.

As a result, the practice of requirements engineering, which deals with processes to generate and maintain the software requirements throughout the duration of the software life cycle, is receiving increased attention. Requirements volatility factors are now considered to be of paramount importance in order to assess critical project success factors such as cost, schedule, and quality.

For example, our COCOMO II research [Boehm et al. 2000] led us to introduce a factor for Requirements Evolution and Volatility (REVL) to adjust the effective size of the product caused by requirements evolution and volatility. REVL is the percentage of code discarded due to requirements evolution from any sources. With a larger effective size in COCOMO due to REVL, the effort includes the extra work due to requirements volatility and not just the work associated with the final kept software.

In [Pfahl, Lebsanft 2000] a simulation model demonstrated the impact of unstable software requirements on project duration and effort. It allowed one to determine how much should be invested in stabilizing requirements for optimal cost effectiveness. Another model for requirements volatility was developed in [Ferreira et al. 2002] and is described next.

5.9.1 Example: Software Project Management Simulator

An effort to model and illustrate the far reaching effects of requirements volatility on the software development lifecycle was carried out at Arizona State University by Su-

san Ferreira, James Collofello and colleagues in [Ferreira et al. 2003]. This initiative is of particular interest because previous requirements volatility models were specific to an organization or had limited inclusions of requirements volatility and requirements generation. Table 5.11 is a high-level summary of the model with different work phases abstracted out.

5.9.1.1 Background

The Software Project Management Simulator (SPMS) model is targeted at a software development project manager or a manager who wishes to better understand the effects of requirements engineering and requirements volatility on critical project factors such as cost, quality, and schedule. The model is relatively complex and assumes that the user has prior knowledge of simulation models. The model relies on a significant and proven foundation of software project management and simulation research.

Development of the model was started with an extensive review of related literature on the subject. Before the model was developed, however, requirements engineering process model and workflows, an information model, and a causal model were developed in order to better understand the involved relationships.

Table 5.11. Requirements volatility model overview (abstracted)

Purpose: Planning, Process Improvement
Scope: Development Project

Inputs and Parameters	Levels	Major Relationships	Outputs
• Change requests • Change request acceptance rates • Requirements volatility rates • Staffing profiles • Productivity rates • Quality assurance effectiveness • Rework policies	Requirements sector only: • Requirements Undiscovered Additional To be worked Generated Awaiting review Reviewed With errors With errors reworked • Requirements product with defects With defects reworked • Requirements changes To be reworked Generated Awaiting review Reviewed With errors With errors reworked	• Schedule pressure effects on error generation • Process delays • Productivity functions • Review backlog tolerances	• Effort • Schedule • Defects

An information model, which covered part of the model's context, was developed in order to gain further knowledge about requirements and change requests prior to modeling. The identification of relevant factor relationships led to the development of the causal model shown in Figure 5.52. Factors such as the review of relevant literature and discussions with various software engineering experts contributed greatly to the development of the causal model.

Concurrently, a use case along with various user scenarios were created to make the model development more focused and to serve as a training tool for others. These scenarios also helped identify additional elements that needed to be added to the model and to check parameter coverage.

A number of walk-throughs and reviews were held with selected researchers. The model was modified to incorporate the key suggestions of the reviewers and the model was then considered ready to receive quantitative data.

In order to populate the model with relevant data, a survey was developed and conducted. The survey was used to obtain data for only a subset of the causal model factors as relevant data for some relationships were already available. The survey was sponsored by the Project Management Institute Information Systems Specific Interest Group. In addition, the survey also targeted the members of the SEI Software Process Improvement Network (SPIN) and individual professional contacts. Over 300 software project managers and other development personnel responded to the survey, and 87% of them were project managers or had some kind of a lead role.

The survey showed that 78% of the respondents experienced some level of requirements volatility on their project. The survey showed that the effects of requirements volatility extended to:

- Increase of job size
- Major rework
- Morale degradation
- Increase in schedule pressure
- Decrease in productivity
- Increase in requirements error generation

The histogram in Figure 5.53 indicates the percent change in project size with the corresponding number of projects (frequency). Note the positive skew of the distribution, indicating that, in almost all cases, requirements volatility serves only to increase the project size. On an average basis, survey results showed 32.4% requirements-volatility-related job size increase.

Figure 5.54 shows the distribution of the requirements volatility job size change of a typical project across 10 equal time intervals. The figure clearly demonstrates how requirements volatility is a factor that needs to be considered throughout the entire phase of the project development life cycle. Greater detail on the survey findings showing primary and secondary effects of requirements volatility are discussed in [Ferreira et al. 2002].

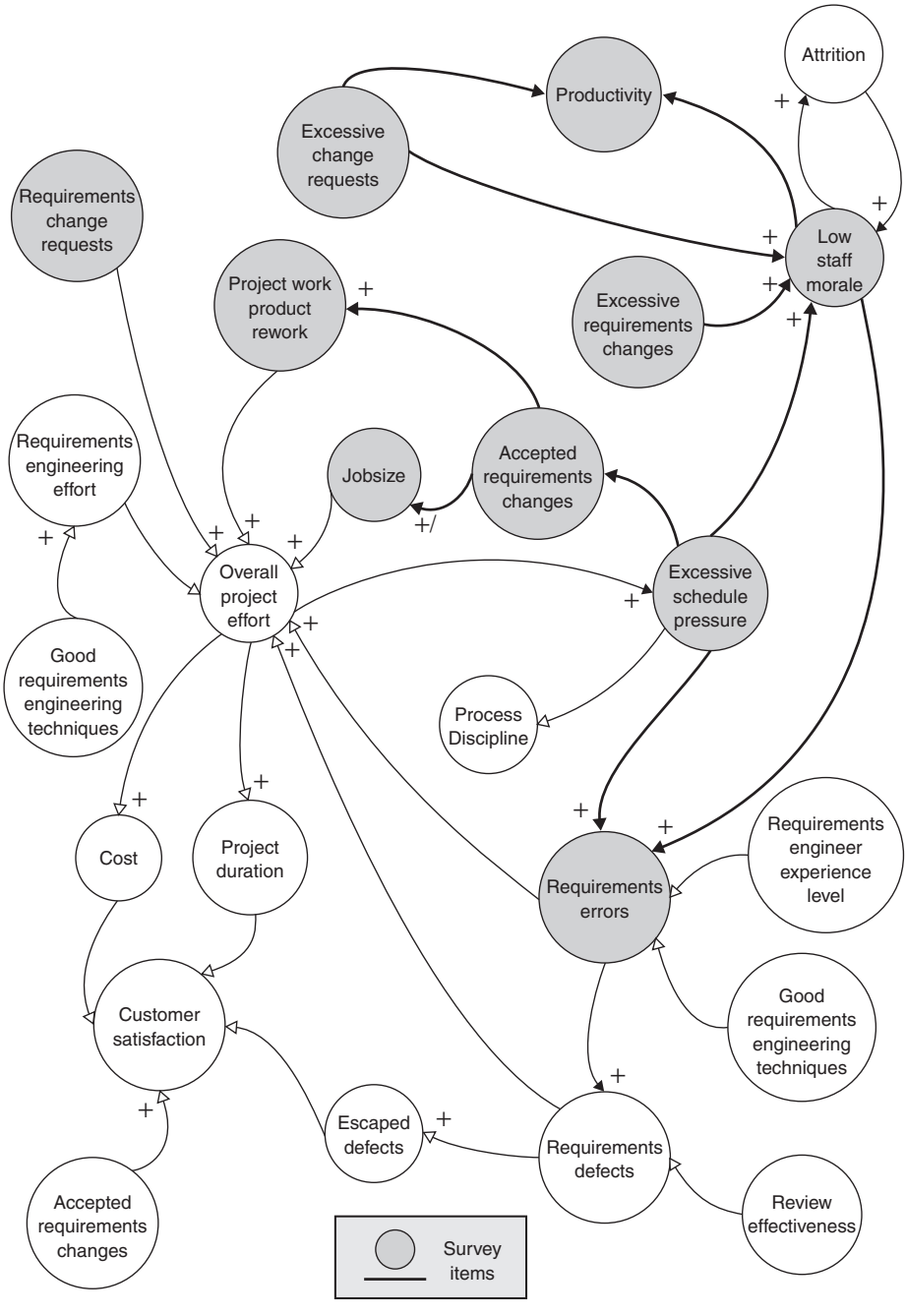


Figure 5.52. Causal model.

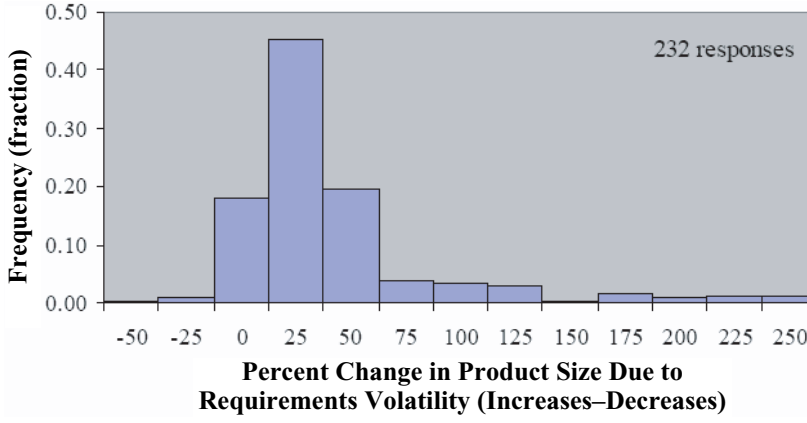


Figure 5.53. Distribution of requirements volatility percent change in product size.

5.9.1.2 Simulation Model

The SPMS was evolved from several previous models. Dan Houston reused and modified the Abdel-Hamid model [Abdel-Hamid, Madnick 1991], integrated it with John Tvedt's dissertation model [Tvedt 1996], and added risk-based extensions to create the SPARS model [Houston 2000]. The SPARS model was further enhanced with requirements engineering extensions and requirements volatility effects to create SPMS.

SPMS facilitates the addition of the requirements engineering phase through the test phase of the software development life cycle, and the requirements volatility effects through the development and test phase of the life cycle. The model also covers change requests, change request analysis, review activities, and their dispositions.

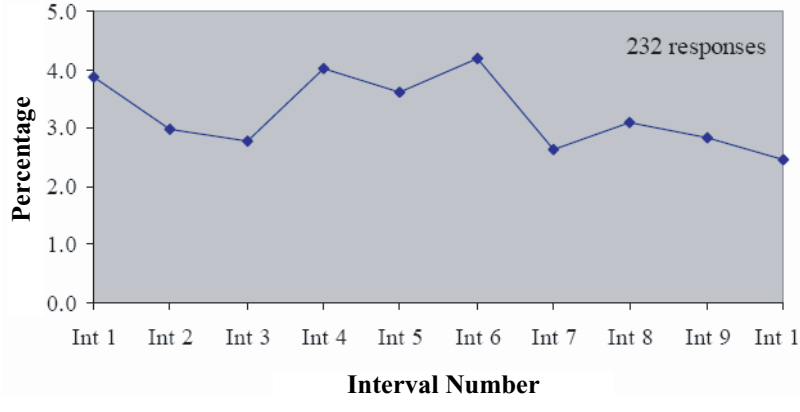


Figure 5.54. Requirements volatility job size change averages, per time interval.

SPMS validates the causal model illustrated in Figure 5.52. Impact of requirements volatility on project size, rework, staff morale, and so on were found to be in agreement with the relationships in the causal model.

The SPMS model included over 50 new distributions that were drawn from the survey data. In addition, a number of distributions were reused from other sources and, in certain cases, parameters were modeled using user modifiable single point inputs. The model also included provisions for:

- Effects of lowered morale on requirements engineering productivity and requirements
- Schedule pressure effects on requirements error generation
- Requirements defect detection effectiveness effects for software development milestones
- Relative work rates of requirements-volatility-related activities compared to their normal work rates
- Addition of requirements engineering staff type
- Addition of requirements engineering support activities

The model is segmented into 20 sectors. Figure 5.55 provides an overview of the requirements engineering work flow sector. The connections on the right of the sector are bidirectional, connecting to the development and test work flow sectors. In the requirements work flow sector, any product in which errors are detected is removed and reworked before flowing into the development and test sectors. In addition, requirements defects caught at later stages also come back to this sector to be reworked. The reworked product then flows back into the development and test sectors. Additions to job size due to volatility and underestimation flow into the normal work flow through the course of the project.

The lower half of the sector includes the requirements change rework flows and contains separate requirements-error-related activities. Rework due to requirements volatility is taken from the development and test work flows and is worked though the requirements change rework work flow.

5.9.1.3 Simulator Results

The SPMS model was populated with a SPARS base case for testing. The SPMS model results were then compared to check if differences existed. The assumption was that the SPMS model with the requirements volatility risk-based extensions removed should give approximately the same results as the SPARS model. Cost and schedule outputs of the executed test scenario for 64 thousand lines of code (KSLOC) are in Table 5.12. As is shown in the table, the time difference is very small (< 1 day). The cost in terms of person days is within 7% of the SPARS base case.

There are a number of reasons that help explain the difference in the values. The SPMS model uses function points as inputs whereas the SPARS model uses KSLOC based units. Hence, in order for a valid comparison to be made, the function points

were converted to equivalent KSLOC values using a language conversion factor and a backfiring value, which may not always be a 100% accurate. Certain input factors in SPMS are stochastic, implying that a 100% results match between the values produced by SPMS and SPARS will not always be possible. In addition, the SPMS model has certain extensions which provide for recognition of requirements defects in later project phases. This is not modeled in the SPARS model.

A separate set of results shows the comparison between the base case runs and the runs with the requirements volatility risk-based extensions actualized. The results are displayed in Figure 5.56 and Figure 5.57.

Box plots are used as the results produced by the test runs are positively skewed. The box plots graphically display the center and the variation of results. As can be seen in both Figure 5.56 and Figure 5.57, the baseline results also show some degree of probability due to the stochastic nature of inputs. This implies that probabilistic results may appear, even with the requirements-volatility-based extensions turned off. The large variation in outcomes for both cost and schedule clearly demonstrate the potential for unpredictable results due to requirements volatility.

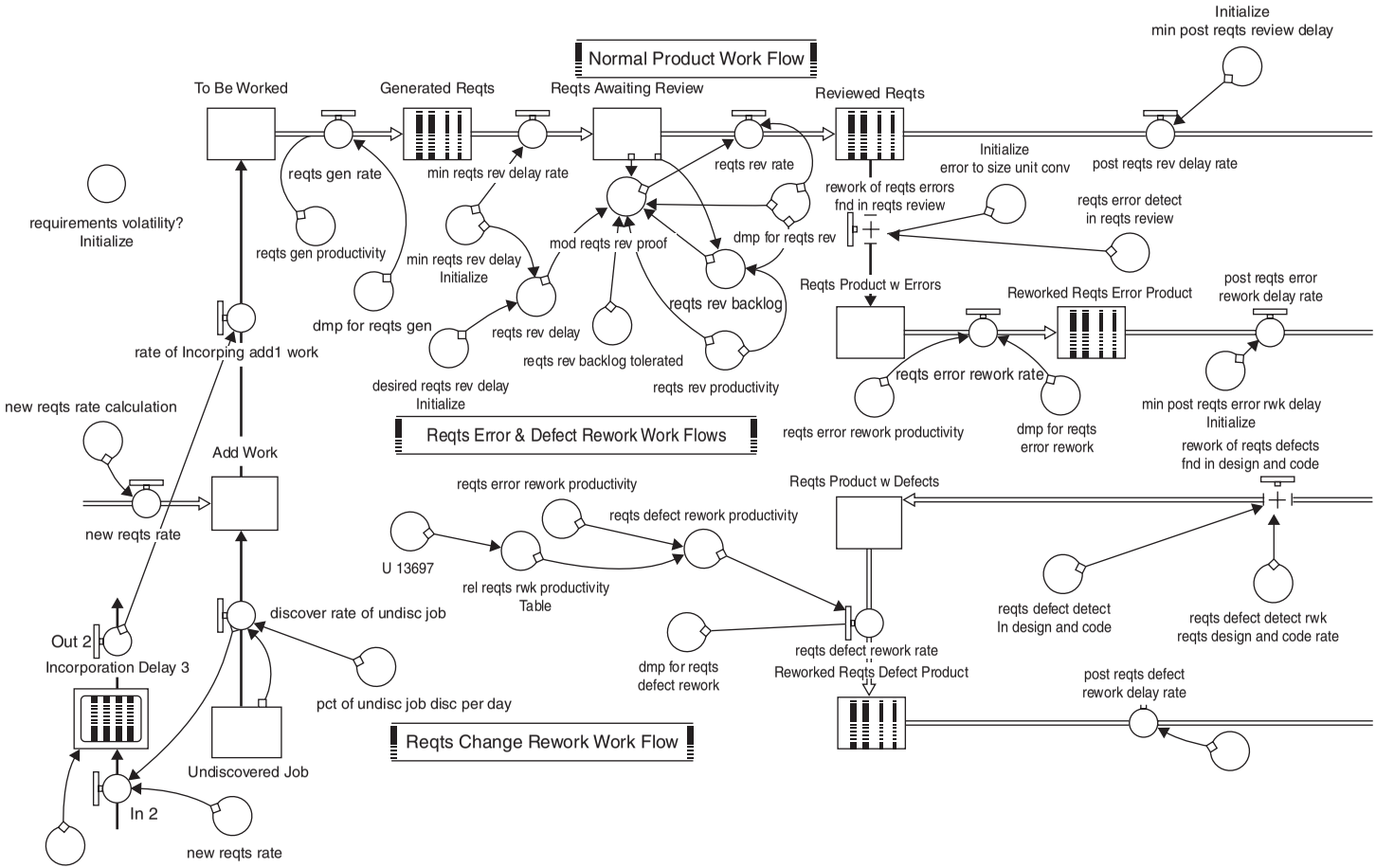
5.9.1.4 Summary

The SPMS model clearly shows the effects of requirements volatility on critical project factors such as cost and schedule. In addition, the simulator can also be used to evaluate potential courses of action to address the risk of requirements volatility. The work helps to better understand the intricate relationships that exist in a software development lifecycle. In addition, the data that was gathered through the survey has helped to fill in important quantitative gaps in project relationships.

5.10 SOFTWARE PROCESS IMPROVEMENT

Software process improvement (SPI) at an organizational level is one of the most important applications in this book. Virtually all companies are (or should be) concerned with continuous process improvement, or they risk languishing behind their competitors. Many companies allocate substantial resources for SPI initiatives and they expect returns. Modeling can frequently give them a better idea of where to put resources and provide insights into organizational policies and expected results. Several SPI frameworks were introduced in Chapter 1. The reader is encouraged to review that section and CMM introductory concepts as background for this section, as necessary.

An important study of process improvement dynamics in a high-maturity organization was commissioned by the SEI and conducted by Steve Burke [Burke 1996]. He developed an extensive model at the Computer Sciences Corporation (CSC) using the NASA Goddard Space Flight Center Software Engineering Lab (NASA SEL) environment to model software development and SPI processes. This report is recommended reading for those interested in using simulation to analyze SPI potential and is reviewed in the example section below.


Normal Product Work Flow
To Be Worked
Generated Reqts
Reqts Awaiting Review
Reviewed Reqts
Initialize
min post reqts review delay
reqts gen rate
min reqts rev delay rate
reqts rev rate
Initialize
error to size unit conv
post reqts rev delay rate
rework of reqts errors
fnd in reqts review
reqts error detect
in reqts review
requirements volatility?
Initialize
reqts gen productivity
min reqts rev delay
Initialize
mod reqts rev proof
dmp for reqts rev
Reqts Product w Errors
Reworked Reqts Error Product
post reqts error
rework delay rate
dmp for reqts gen
reqts rev delay
reqts rev backlog
reqts rev backlog tolerated
rate of Incorping add1 work
desired reqts rev delay
Initialize
reqts rev productivity
reqts error rework rate
new reqts rate calculation
Reqts Error & Defect Rework Work Flows
reqts error rework productivity
dmp for reqts
error rework
min post reqts error rwk delay
Initialize
rework of reqts defects
fnd in design and code
Add Work
reqts error rework productivity
Reqts Product w Defects
reqts defect rework productivity
new reqts rate
U 13697
rel reqts rwk productivity
Table
reqts defect detect
In design and code
reqts defect detect rwk
reqts design and code rate
discover rate of undisc job
Out 2
Incorporation Delay 3
dmp for reqts
defect rework
reqts defect rework rate
Reworked Reqts Defect Product
post reqts defect
rework delay rate
pct of undisc job disc per day
Reqts Change Rework Work Flow
Undiscovered Job
In 2
new reqts rate

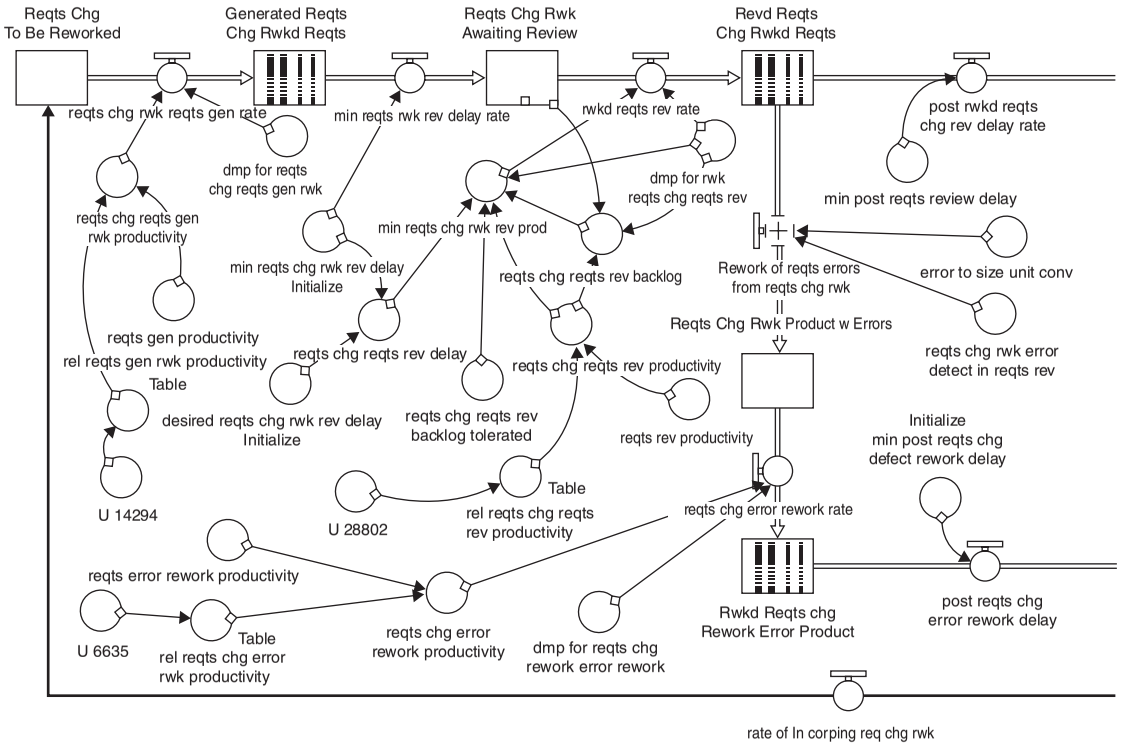


Figure 5.55. SPMS requirements work flow sector sample.

Table 5.12. Test runs comparison between the SPARS and the SPMS models

	Schedule (days)	Cost (person days)
SPARS	383	3606
SPMS	384	3870

Experience with a system dynamics model for studying strategic software process improvement in the automotive industry is described in [Pfahl et al. 2004a]. The collaborative effort with DaimlerChrysler AG aimed to explore the level of process leadership to be achieved by one of the divisions. By identifying and determining the influence factors and expected measures of process leadership, a generic modeling structure was developed that can serve as a framework for other companies examining improvement strategies.

5.10.1 Example: Software Process Improvement Model

The original goal in [Burke 1996] was to find the quantitative value of improving from CMM Level 3 to Level 5. However, NASA SEL uses a different process improvement mechanism than the CMM, called the "Experience Factory." The slightly refined goal was to determine the impact of various management-level policy decisions. The resulting model shows the value of high maturity in general, the value of the Experience Factory in particular, and how the Experience Factory can relate to the CMM in a very generalized sense. As discussed later and used in the Xerox adaption study, there is a

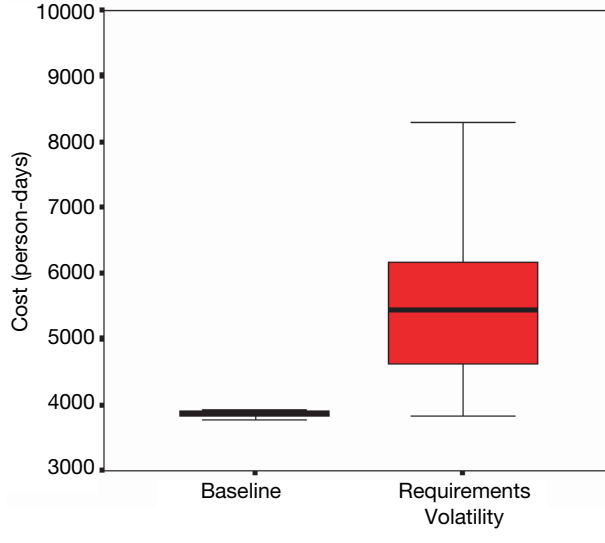


Figure 5.56. Project cost box plot.

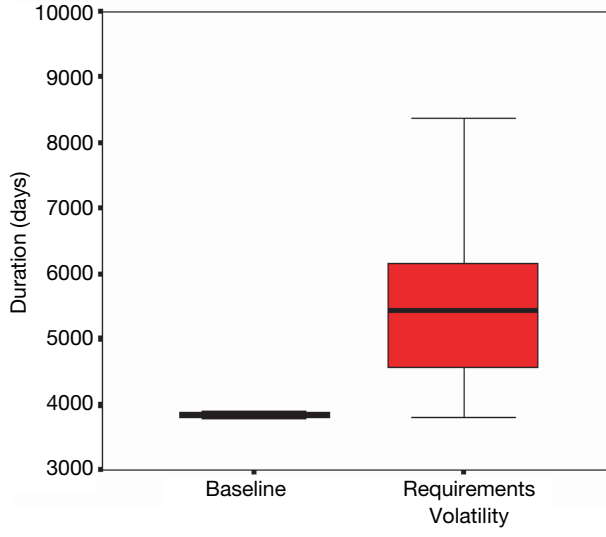


Figure 5.57. Project schedule box plot.

fair equivalence between Key Process Areas (KPAs) and "Major SPI suggestions" in the model. The latter are used as surrogates for KPAs.

The model was used by a number of major companies to evaluate process improvement options. This author was briefed on a major effort undertaken by Hughes Spacecraft and Communications (now Raytheon) to adapt the model internally and perform extensive sensitivity analyses. The Xerox adaptation of the model and their experience is covered later in this section. Table 5.13 is a high-level summary of the model.

5.10.1.1 High-Level Feedback

Figure 5.58 shows a high-level diagram of the model and the key feedback relationships. The basis of the simulation representing the software development and process improvement processes used by the NASA SEL is described in the following causal loop:

1. Major software process improvement (SPI) efforts are piloted and deployed based on project cycle time (i.e., piloted on one project, tailored and deployed on the subsequent project).
2. Major SPIs increase maturity (the probability of successfully achieving your goals).
3. Increased maturity attracts new hires and retains experienced staff that are "pro SPI" (they support and participate in SPI activities and are attracted to success and innovation).
4. Pro-SPI staff makes minor SPI suggestions.
5. Major and minor SPIs decrease cycle time.

Table 5.13. Software process improvement model overview

Purpose: Process Improvement, Planning
Scope: Multiple Concurrent Projects, Long-term Organization

Inputs and Parameters	Levels	Major Relationships	Outputs
• Size • Initial workforce • Pro/con ratio • Cutout flag • Initial major SPIs • Suggestion per pro per cycle • Minor ROI improvement • Percent of suggestions from outside • Active hiring • Quit rate	• Software process improvements Initial major Approved major Piloted major Deployed major • Personnel New hires Pros in training Pros experienced Cons in training Cons experienced No-cares in training No-cares experienced • SPI size • SPI schedule • SPI effort • SPI errors	• SPI deployment rates • SPI adoption rate • People attitude transfer • Hiring policies • Error rates • Delays	• KPAs by phase • Effort • Schedule • Errors • Staffing by type • SPI major maturity • SPI minor maturity

6. Decreased cycle time enables more major and minor SPIs to be accomplished.
7. Go back to 1 and repeat the cycle.

This high-level feedback is further elaborated in the model and its subsystems are described next. The top-level mapping layer of the model in Figure 5.59 shows "process frame" building blocks. These are the important aspects of the system that can be drilled into. The arrows linking the process frames in the diagram show the relationships between the various pieces of the model.

Control Panel I in Figure 5.60 is the "dashboard" or "cockpit" from which to perform flight simulation test runs. This control panel is the major user interaction area for this particular study, which shows the following parameters available for change via knob inputs:

- Size of the project in KSLOC
- Initial organization pro/con ratio
- Cutout or Noncutout

It also shows a graph output to display cumulative major SPIs being piloted or deployed as the model runs in real time. Numeric display bars show outputs such as improved size and schedule numbers and others.

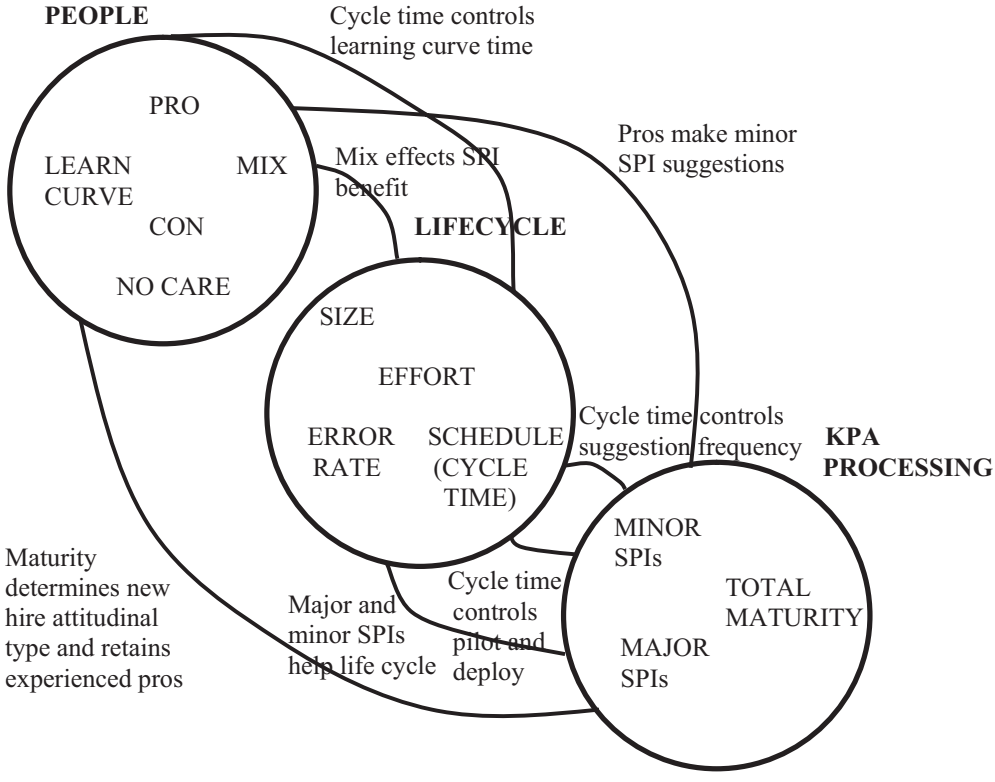


Figure 5.58. High-level diagram of NASA SEL feedback relationships.

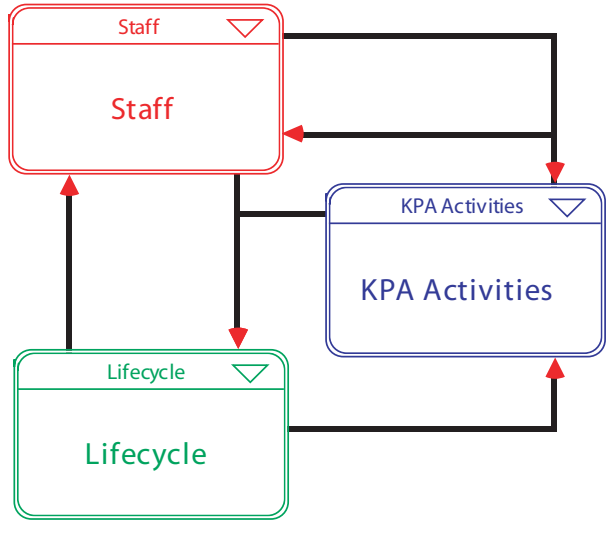


Figure 5.59. Model high-level diagram.

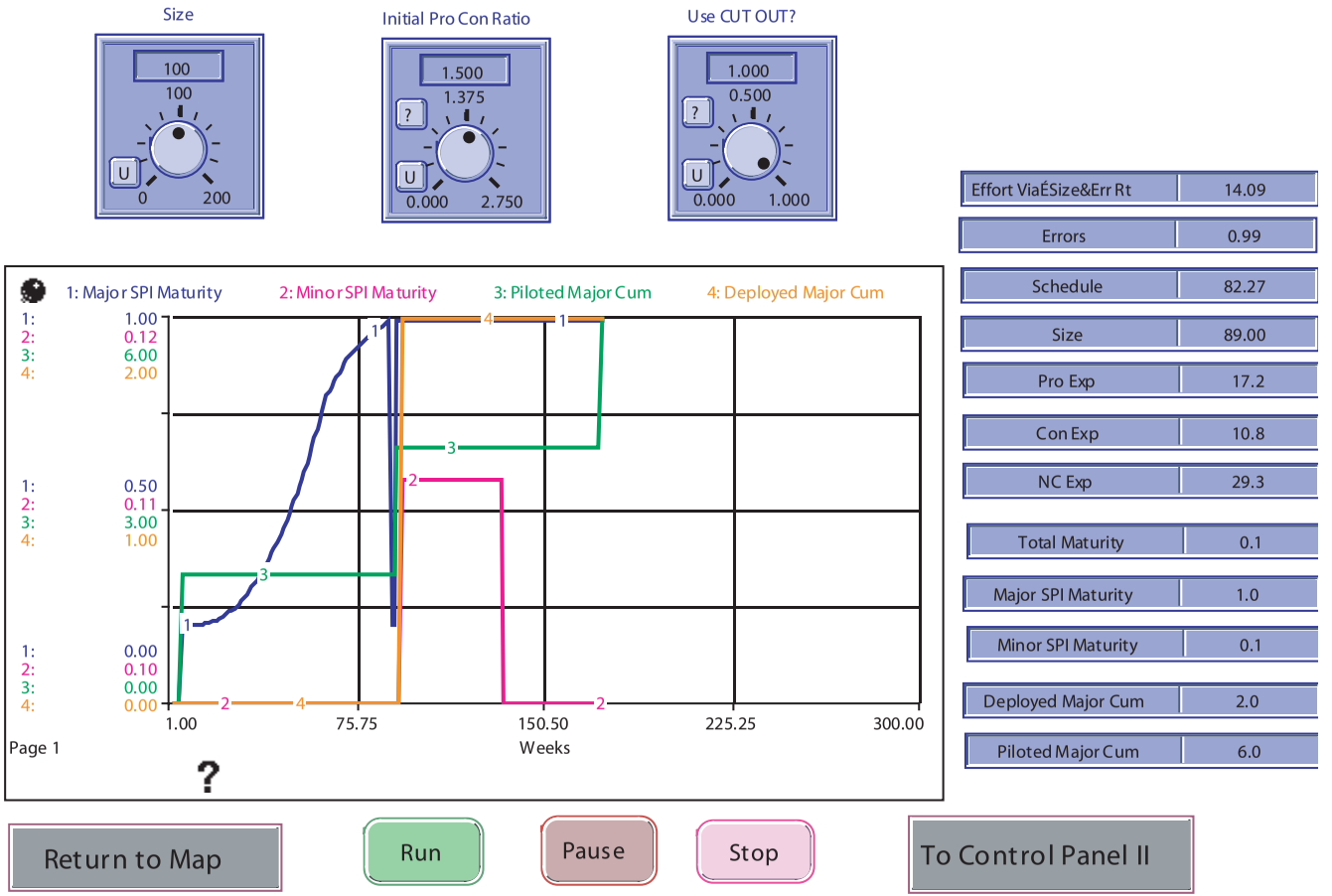


Figure 5.60. Control Panel I.

Control Panel II in Figure 5.61 groups various input parameters used by different sections of the model, and enables easy adjustment of the parameters for test runs. The model subsystems are described next.

5.10.1.2 Life-Cycle Subsystem

The life-cycle process model shows how software size, effort, quality, and schedule relate to each other in order to produce a product. They could be changed by process improvements or by each other as one process improvement rippled through the four at-

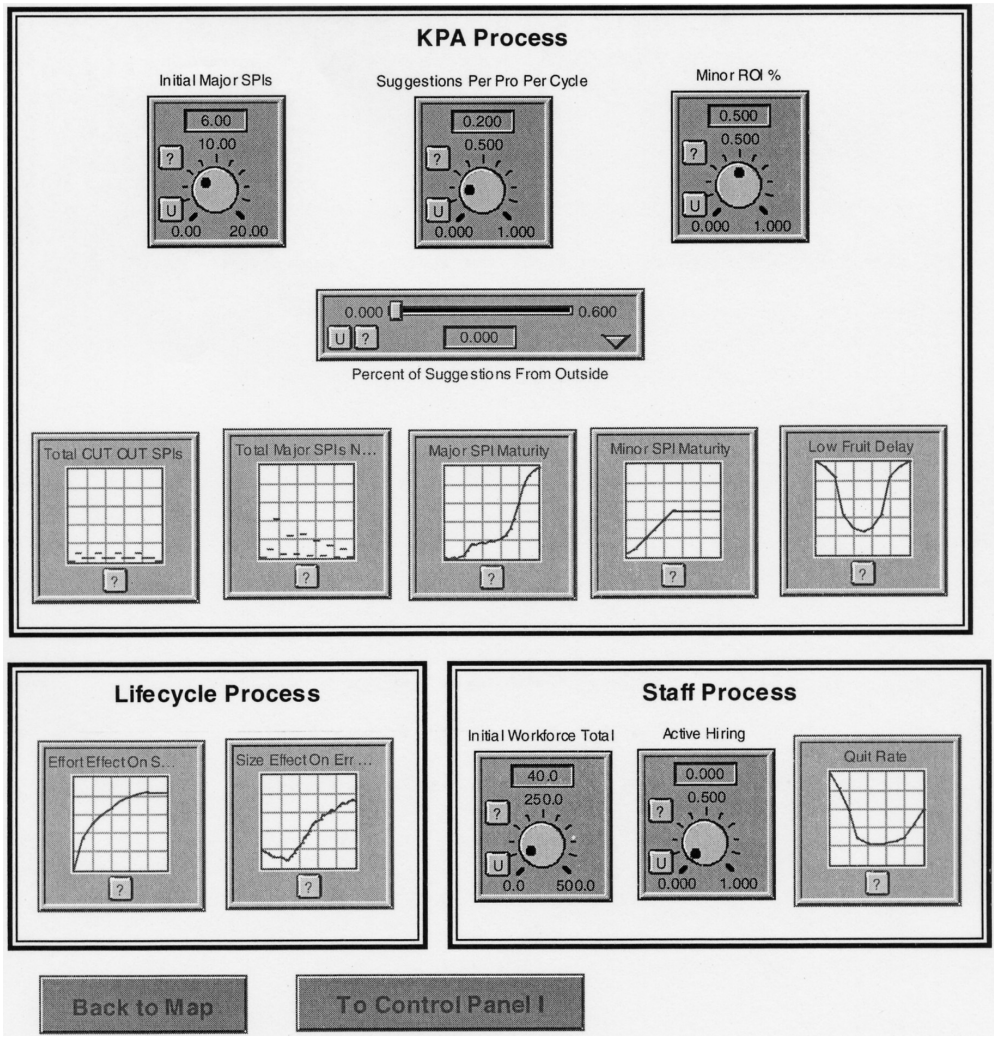


Figure 5.61. Control Panel II.

tributes. SPI benefits are modeled as percent reductions in size, effort, error rate, or schedule.

The life-cycle subsystem is shown in Figure 5.62. Starting at the upper left-hand corner and working clockwise, the personality (attitude) mix converter adjusts any of the schedule, size, ROI rates based on the pro-to-con ratio. Moving over to the top center, cumulative size SPIs affect the rate of change of size. In the upper-right corner, size changes affect error rate according to a defined graph. Also, an SPI may affect the error rate directly. In the lower-right corner, the impact that size or error rate might have on effort is modeled. In the lower-center part of the diagram, changes in effort due to size or error rate changes are combined with changes in effort due to effort SPIs to model overall effort changes. Finally, in the lower-left corner, effort changes affect schedule according to a graph. Schedule SPIs may also affect schedule directly.

5.10.1.3 People Subsystem

The model simulates three attitudes of project staff that affect the potential benefit of process improvement: pro-SPI people, con-SPI people, and no-care people. One of the important lessons of the model is that organizations should pay attention to the attitudinal mix of personnel. Each of the types are represented as levels. The attitudinal mix and the pro/con ratio can affect the overall potential benefit realized by an SPI effort. This was defined as the attitude impact. It also assumed that it takes one project cycle for a new hire to become experienced. The project cycle time is the learning curve.

If there are more pros than cons, then more no-cares will also adopt a process improvement effort. This higher penetration and adoption will realize a higher overall benefit. If there are a lot of cons, then a lot of people may not adopt the SPI effort. Interviewees agreed that attitude affected SPI adoption and that staff members with strong attitudes affected other staff members' adoption of SPI.

A subjective survey on an Ada/reuse major SPI effort showed about 30% of the staff said they were pro-Ada, 20% were con-Ada, and 50% did not care. Both the pros and the cons were vocal in their opinions about this major improvement effort. The following "soft" variable relationship between pro/con ratio and personality mix effectiveness was derived using the 30%, 20%, and 50% values:

1. At a 30/20 ratio, assume the 50% no-cares go along with the pros.
2. Thus, 80% of the total staff are virtual pros. Since this was the documented case, their personality mix effectiveness was set to 1. That is, if an SPI effort claims to give a benefit of 50%, then the personality mix effectiveness given this personality mix is 50% times 1, or 50%.
3. To find the effectiveness for a 25/25 pro/con ratio, the following is used: Assume the no-cares are also split, since the pro/con ratio was even. If 80% virtual pros produced an effectiveness of 1, then having 50% virtual pros produces an effectiveness of 50%/80% = 0.625.
4. If there is a 20/30 pro/con ratio, assume the no-cares are virtual cons. So having only 20% pros produces an effectiveness of 20/80 = 0.25.

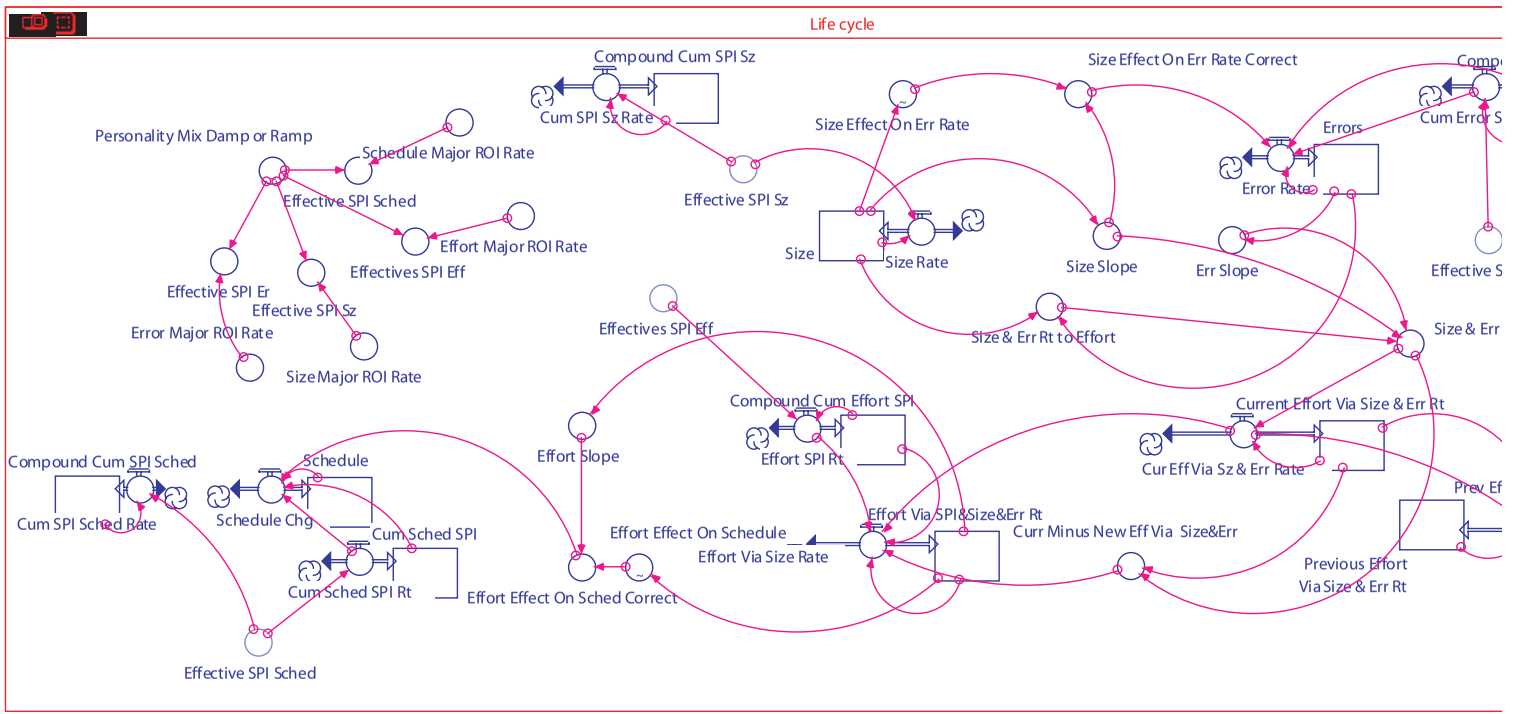


Figure 5.62. Life-cycle subsystem.

Referring to Figure 5.60 and starting at the right, experienced no-cares, cons and pros are separate flows that quit according to the quit rate. The quit rate itself changes with maturity. For each quit, a no-care, con, or pro new hire is determined by a Monte Carlo probability distribution that is also based on maturity (see the left part of the diagram). If an experienced con quits, a pro may replace him or her if the maturity is high. The time the no-cares, cons, or pros spend in training (the middle slotted boxes) is regulated by the current project schedule. As process improvements decrease cycle time, the learning curve time is also decreased. As process improvements increase maturity, the attitudinal mix based on the new pro and con totals also changes because new hires may have a different attitude than the experienced person who quit.

5.10.1.4 KPA Processing Subsystem

The KPA processing subsystem is the most complex subsystem. It models the timing of the flow of process improvements into the life cycle and people subsystems. There are two types of SPIs in Burke's model: major and minor. This maturity variable does not directly map to the five levels of the CMM, but uses adoption of SPIs and suggestions as surrogates for maturity levels. The total maturity is defined as being a composite of major SPI maturity and minor SPI maturity. Each of these was set on a scale of zero to one.

Figure 5.64 shows the KPA processing subsystem. The top half deals with major SPIs, and the bottom half models the minor SPI suggestion process. For the major SPI section starting from the left, major SPIs are piloted and deployed as regulated by schedule (see top center for the schedule box). In the center, the "Pilot Major Cum" and "Deployed Major Cum" receive the impulse of the completed piloted or deployed SPI at the end of a cycle and store the cumulative total. In the center, the "Total Major SPI" sends out the appropriate ROI percent based on when the pilots and deploys are complete. This ROI percent is sent out to one of either the "Size Major ROI," "Effort Major ROI," "Error Major ROI," or "Schedule Major ROI" flows located in the top-right corner.

For the bottom half that models the minor SPI suggestions, the bottom center shows the "Sug Frequency" being a product of "Total Maturity," number of pros, "Suggestions per Pro per cycle," and "Percent of Suggestions from Outside." The structure to the right of this causes the suggestions to be implemented in a "low-hanging fruit" fashion. That is, within a project cycle, a few suggestions are made at the beginning, many in the middle, then only a few at the end since all of the easy ones (the low-hanging fruit) were worked. The structure in the lower-left corner is needed to compute the average suggestion frequency for a floating period of one-half of a project cycle. As the suggestion frequency increases, the "Minor SPI Maturity" also increases because when an organization is mature, everyone is involved in making improvement suggestions.

5.10.2 Example: Xerox Adaptation

The Xerox Corporation adapted the Burke SPI model to support planning, tracking, and prediction of software process improvement activities for their Common Client & Server Development (CCSD) group. Jason Ho led the effort to model the scenario of

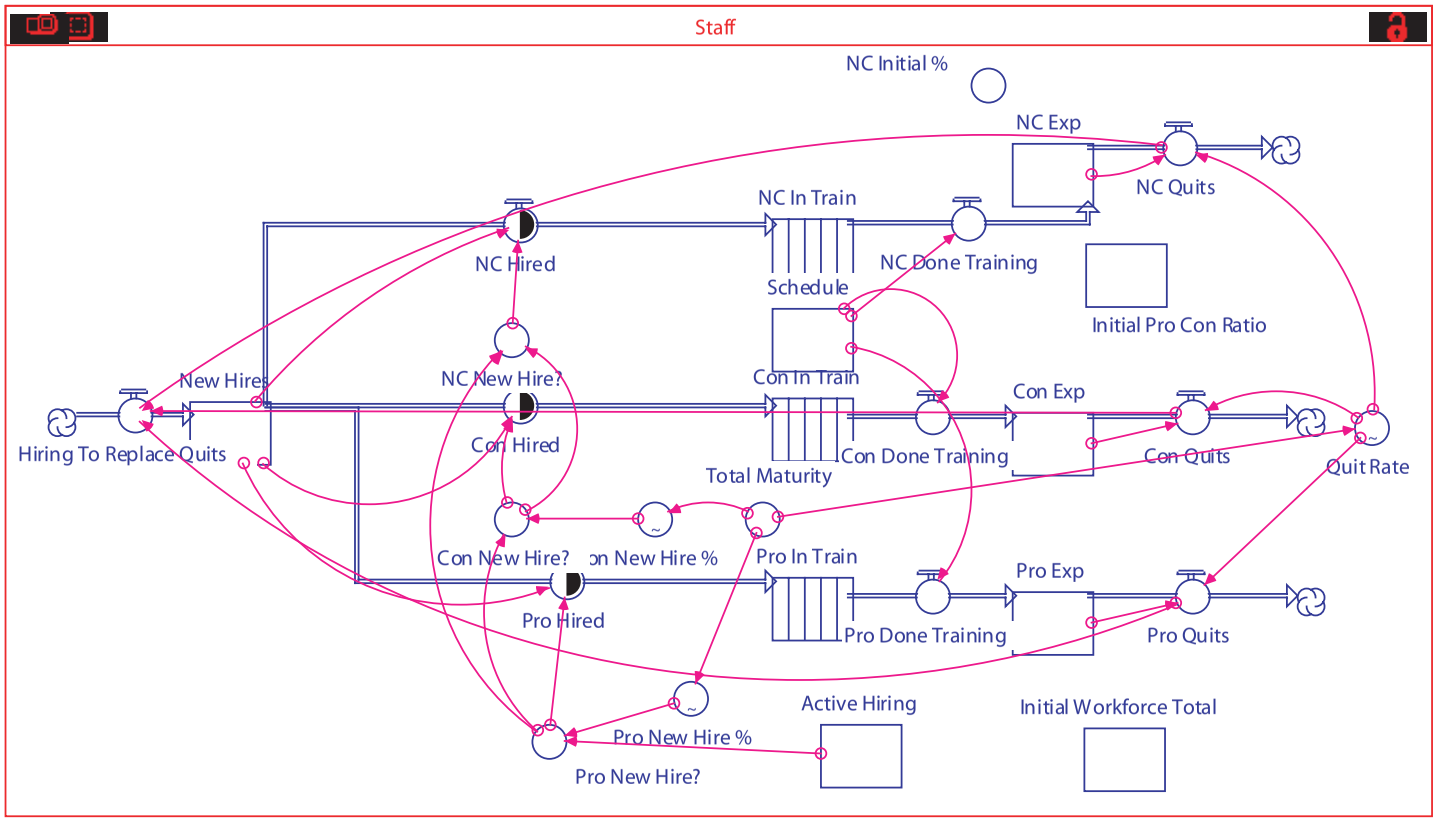


Figure 5.63. People subsystem.

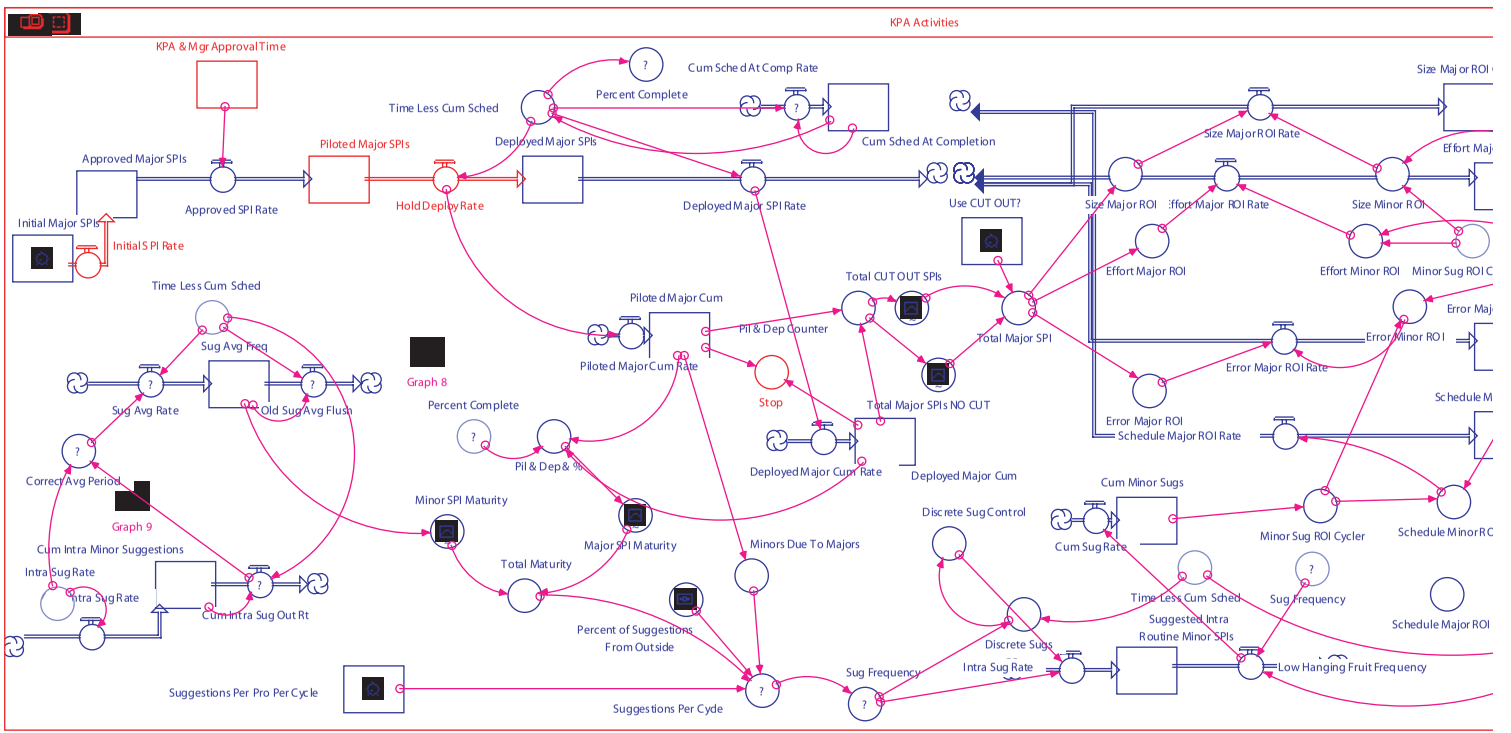


Figure 5.64. KPA processing subsystem.

the group working toward achieving CMM Level 3 after recently being assessed as a CMM Level 2 organization. Challenges to resolve in adopting Burke's model included modeling the Experience Factory versus the CMM and handling two types of process improvements. The original model had major and minor improvements versus the notion of KPAs in the CMM.

In order to reuse Burke's model for simulating Xerox software group's CMM SPI activities, the following assumptions and modifications were made:

- The basic model structure reflects a general software development organization's product life-cycle and process improvement behaviors, though it was specifically simulating NASA GSFS SEL with its own data.
- Xerox software organization data was used to calibrate the life cycle and the people processes replacing the calibration to SEL data.
- The definition of major SPIs in Burke's model can be equated to KPAs in CMM model.

Complete details can be found in [Ho 1999]. Following are highlights from that report. Changes were made to the subsystem models and the user interface, and new test cases were developed. Specific adaptations to the model sectors and their parameters are described next.

5.10.2.1 Model Changes

The life-cycle subsystem was left as is but a new calibration was desired. Initially, there was no data available from Xerox to recalibrate the percentage (reduced) effects for measuring the benefit of CMM KPAs to project life-cycle elements (size, effort, error rate, and schedule) in the model. NASA data on major SPI benefits was adopted for reference instead, until Xerox data became available.

The people subsystem was evaluated for relevance to Xerox. The people attitude phenomenon in the model was personally experienced within the Xerox software development group. Quick polls of the Xerox group under study showed that the nominal case of a 30/20 pro–con ratio and 50% no-cares seemed to hold true for the organization. This was used to set the attitude proportions for the Xerox environment.

The SEL reported that turnover started at about 10% per year, then decreased to 2–3% as maturity increased, then increased to about 6% toward the end of the 10-year period. For the previous five years, the turnover rate at the Xerox group under study had been flat at 5% year by year, regardless of process maturity. In the test run, "quit rate" was modified to reflect Xerox's situation.

The KPA processing model area was the major focus of the study for Xerox purposes. Given the assumption that the definition of major SPIs in Burke's model can be equated to KPAs in a CMM model, major SPI and CMM KPA can be used interchangeably in the model. There are seven major SPIs (seven Level 3 KPAs in the CMM) to be piloted and deployed by projects in order to reach the next level of maturity. Based on CMM assessment criteria, all KPAs must be implemented (piloted), plus some need to be repeated (deployed) to satisfy the institutionalization require-

ments. Thus, the pilot/deploy scenario simulated in Burke's model can be adopted to reflect the CMM assessment requirements.

Burke's model assumes that one can only pilot or deploy one major SPI per cycle (i.e., a project). Test runs verified that it will take too long to implement all seven CMM KPAs with the original model. Validated by Xerox CCSD's experience on achieving CMM Level 2, two KPAs can be implemented per project on average. To simulate the average of two KPAs rolled out per project, the rate of the following elements in the model (as shown in Figure 5.65) was changed from one to two for the Xerox CCSD nominal case:

- Initial SPI Rate
- KPA & Mgr Approval Time
- Piloted Major SPIs
- Hold Deploy Rate

Also, in order to capture the simulated lapsed time in implementing seven Level 3 KPAs, an element needs to be added to stop the model running after seven major SPI roll outs. In this particular Xerox CCSD case, the already implemented Level 3 activities is assessed as worth one full KPA. So the modification is to stop the model after six major SPIs have been piloted. Figure 5.66 shows the element added.

One of the major issues faced by Xerox was that there was no group of full time process people to help with process improvement activities, so the group had to use the same resources (people who did software development) to work on process related activities, such as to define an organizational process and plan for the roll out. In a general sense, this will prolong the process journey in reaching the next level of maturity on the CMM ladder. Moreover, in many cases, it is not the general approach taken by other major companies working on their CMM SPI. There is a similar condition simulated in Burke's model, called cutout versus noncutout cases.

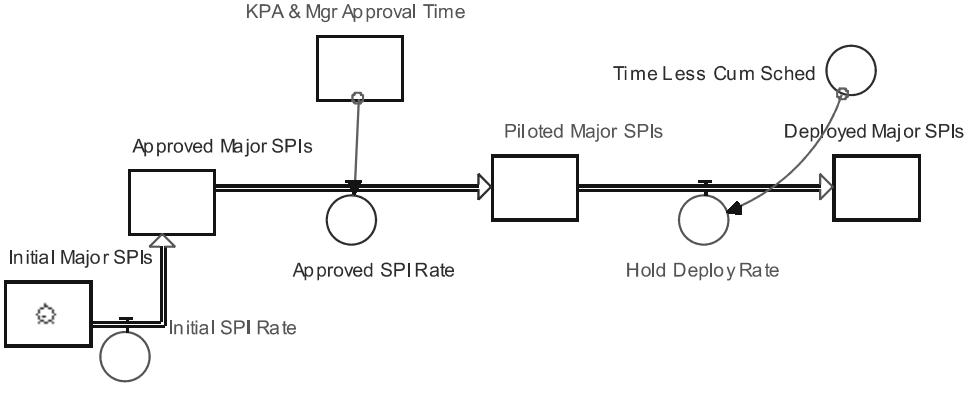


Figure 5.65. Changed KPA processing elements.

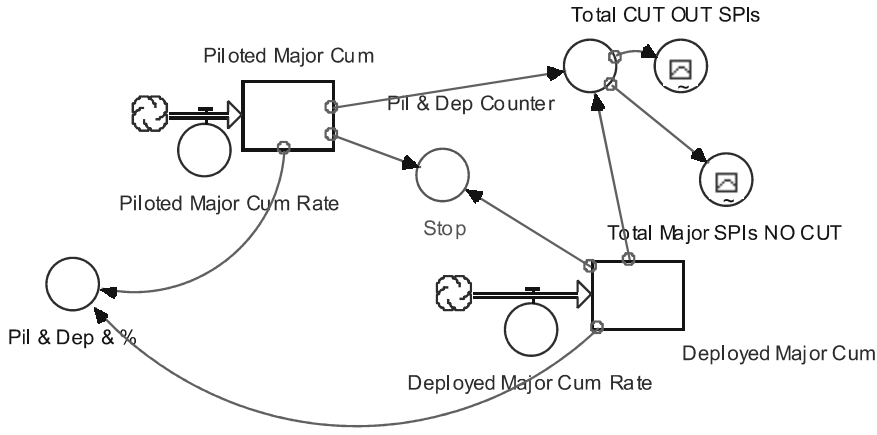


Figure 5.66. Added KPA processing elements.

In the cutout cases, projects accomplish SPI activities in isolation with no sharing of lessons. The noncutout case has a fully staffed group that would accomplish organization-level SPI activities across projects using the CMM Maturity Levels 4 and 5 key process area activities as a guide to pilot and deploy major SPIs across projects, extract lessons from projects, communicate them to other projects, and staff an organization-level metrics program. The SEL calls this staff the "Experience Factory" staff.

The Xerox case is a similar situation of the cutout case, which is the nominal case for the modified model. In addition, the comparison with the noncutout case would warrant the attention of management for the approach on SPI.

5.10.2.2 Test Runs

Test runs with the original and modified model are described here. Two test runs with the original Burke's model are described first, and then six runs that encapsulate Xerox data and calibrations.

5.10.2.2.1 ORIGINAL MODEL. The first test run was based on Burke's baseline case for NASA SEL. Values for the following variables were set: pro/con ratio at 1.5, project size at 100 KSLOC, and noncutout. The initial task force was changed to 40 people and the noncutout was changed to cutout, to reflect Xerox's condition. Results showed the group would not reach next level of maturity in the 10 years (500 weeks) time specified.

Based on the result from the first test run, the initial task force was changed to 100 people in the second run. This represents whether the management decided to grow the group because of extra process burden added. The run showed that the group will reach the next level of maturity in 430 weeks (about 8.3 years).

5.10.2.2.1.1 Modified Model. With the Xerox modified model, the initial task force is set at 40 (the group size of Xerox CCSD). Test runs were performed by chang-

ing the following input variables: project size, initial pro/con ratio, and cutout or noncutout. Table 5.14 describes the six test runs and Table 5.15 shows a summary of the test results against the modified model for the cases.

5.10.2.3 Sensitivity Analysis

Given the model modifications, the purpose of a sensitivity analysis was to find a proper pattern of time progress for achieving a certain number of major SPIs (KPAs). Using the existing Xerox CCSD data as validation, it took four years to move from Level 1 to Level 2 (accomplishing six KPAs).

The parameter "Hold Deploy Rate" determines "Piloted Major Cum Rate," and is the factor that controls the stock "Piloted Major Cum"; see Figure 5.67.

The goal is to find the right major SPI deploy rate to reflect the Xerox CCSD's KPA implementation rate. The nominal case was defined with the following set parameters:

- The size of the project was set at 100 KSLOC
- The cutout versus noncutout flag was set to cutout

Table 5.14. Modified model test run descriptions

Test Run #	Description
1 (nominal Xerox case)	Representing the nominal case for Xerox CCSD, values for parameters were set at 100 KSLOC, 1.5 pro/con ratio, and cutout. Test results show 172 weeks (about three years) for accomplishing six major SPIs (KPAs), with the benefit of final size at 89 KSLOC. This represents the major output of this study—the Xerox CCSD group should make their commitment to the management of accomplishing CMM Level 3 in at least 3 years.
2	All the nominal case variable values stayed the same except that the cutout case was changed to noncutout. Test results show 121 weeks for accomplishing six major SPIs, with a significant benefit on the final size and schedule. This shows that using a different approach with support from a full-time process group would have saved the CMM journey a full year in getting to Level 3.
3	All the nominal case variable values stayed the same except size changed from 100 KSLOC to 200 KSLOC. Test results show 292 weeks for accomplishing six major SPIs. This shows that with a bigger project size, it will take longer to finish the project with the same amount of resources available. It will take longer to pilot out all the KPAs.
4	All the nominal case variable values stayed the same except Size changed from 100 KSLOC to 50 KSLOC. Test results showed 134 weeks for accomplishing six major SPIs. Compared to Test Run 3 with a smaller project size, it will be faster to finish the project and it will shorten the time to pilot all the KPAs. So it make senses to select some small projects for piloting the CMM KPAs, which was a good approach experienced by Xerox CCSD while working from Level 1 to Level 2.
5	Test Run 3 with the noncutout case.
6	Test Run 4 with the noncutout case.

Table 5.15. Xerox test run results.

Input Variables				Output Values				
Test Run #	Size	Pro/Con	Cut/noncut	CMM SCHD	EFF End	ERR End	SCHD End	Size End
1	100	1.5	Cut	172	14	1	82	89
2	100	1.5	Noncut	121	6.5	1	46	56
3	200	1.5	Cut	292	42	3.7	138	178
4	50	1.5	Cut	134	9	1.3	63	45
5	200	1.5	Noncut	180	12.8	1	78	112
6	50	1.5	Noncut	88	4	1	29	28

- The initial pro/con ratio was set to 1.5 (representing 30% pro SPI people, 20% con SPI, and 50% no-care

Several values for "Hold Deploy Rate" were used to see how the stock of "Piloted Major Cum" changes over time. Three values that are reasonable are 1, 2, and 3. The comparative run is shown in Figure 5.68.

The three curves all show the same pattern of step increase. The value to look for is six KPAs at about 208 weeks (4 years). Curve 1 shows six KPAs happen at between 375 to 500 weeks, which is too slow. Curve 3 shows six KPAs happen at between one to 125 weeks, which is too fast. Curve 2 shows it happens between 125 to 250 weeks, which is a reasonable range. So the pilot major SPI deploy rate of two is the value used in the model.

5.10.2.4 Conclusions and Ongoing Work

Results with the modified model satisfied the Xerox need to make the proper schedule commitment for CMM Level 2 to 3. The model helped quantify the duration of process

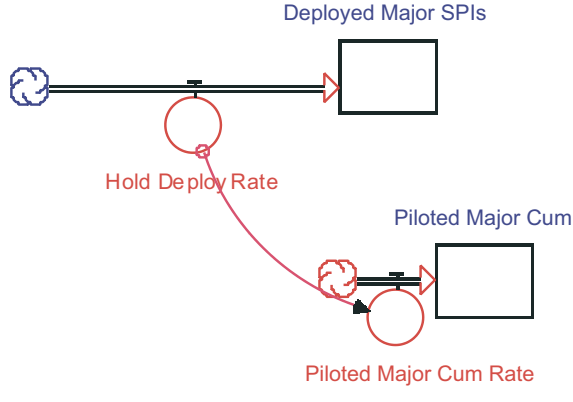


Figure 5.67. SPI accumulation substructure.

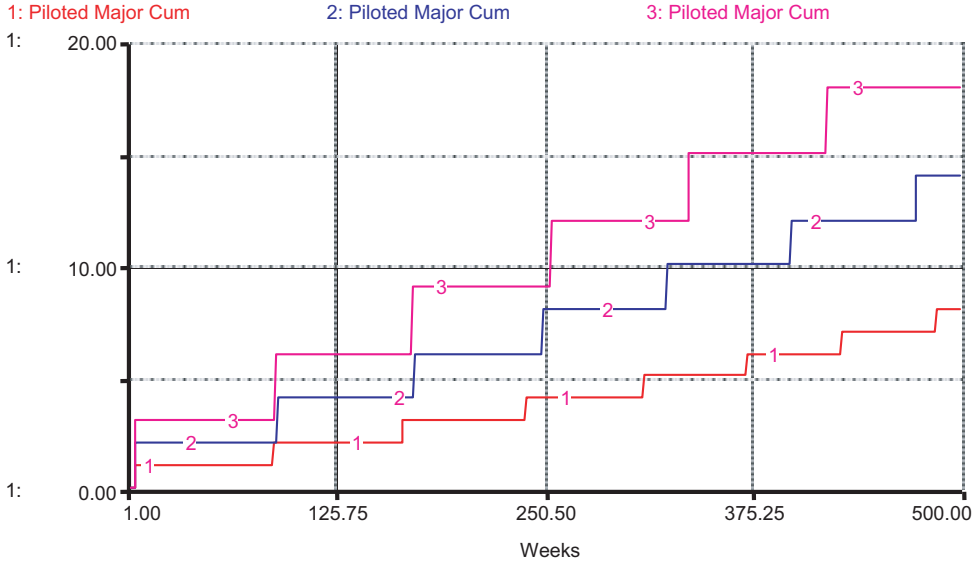


Figure 5.68. Comparative run of "Piloted Major Cum."

improvement and the value of improving software maturity, and brought to light lessons learned by other organizations that have improved. Many of the insights from the model were congruent with previous Xerox experience and data.

One of the most important lessons had to with the attitudinal mix of the software staff. The model showed that having some people with good, proactive attitudes toward process improvement helped instill the same behavior in others. If a critical mass of pro-SPI people are not in place first, then process improvement is much less likely to occur. This lesson is very important to consider when hiring people.

The model will be kept current and recalibrated with more Xerox data as it becomes available. Some specific parameters include recalibrating the KPA benefit percentage with Xerox data, and modifying "Effort Effect on Schedule" with Xerox data. Additionally, the model will be used for minor SPI suggestions and reaching levels beyond CMM Level 3.

5.11 MAJOR REFERENCES

[Abdel-Hamid, Madnick 1991] Abdel-Hamid T. and Madnick S., *Software Project Dynamics,* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Acuña, Juristo 2005] Acuña S. T. and Juristo N. (Eds.), *Software Process Modeling,* New York: Springer Science+Business Media Inc., 2005.

[Kruchten 1998] Kruchten P., *The Rational Unified Process,* Reading, MA: Addison-Wesley, 1998.

[Madachy 1996a] Madachy R., "System Dynamics Modeling of an Inspection-Based Process," in *Proceedings of the Eighteenth International Conference on Software Engineering,* Berlin, Germany, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Paulk et al. 1994] Paulk M., Weber C., Curtis B., and Chrissis M., *The Capability Maturity Model: Guidelines for Improving the Software Process,* Reading, MA: Addison-Wesley, 1994.

[SEI 2003] CMMI (Capability Maturity Model Integration) website, http://www.sei.cmu.edu/cmmi/.

5.12 PROVIDED MODELS

The models referenced in this chapter are provided for use by registered book owners and are listed in Appendix C. See the book website for model updates and additions.

5.13 CHAPTER 5 SUMMARY

Process and product applications cover broad areas and frequently coincide since software artifact quantities and their attributes are primary state variables for both. A majority of system dynamics applications have investigated process issues and not product aspects per se. Part of the reason for this is that other modeling techniques may address product analysis better, so combined approaches would be advantageous.

This chapter discussed some important process and product aspects including peer reviews, software reuse, software evolution, COTS-based systems, incremental and iterative processes, software architecting, quality, requirements volatility, and software process improvement.

Peer review models indicate that extra effort is required during design and coding for the reviews, but they are instrumental in producing higher quality software while also decreasing overall effort and time. Modeling the impact of formal inspections concluded this, and showed that testing effort and schedule were substantially reduced under nominal conditions. However, the cost-effectiveness of inspections varies according to factors like inspection efficiency, defect injection rates, defect amplification, testing, and fixing effort. Peer reviews in general are a good idea, but they should not be overdone as determined by local project conditions.

Software evolution is an iterative process largely driven by feedback as changes are initiated, driven, controlled, and evaluated by feedback interactions. Understanding and modeling global feedback can help address new and emerging trends such as globalization, COTS, or open-source development. A software evolution model for progressive and antiregressive work showed that the optimal level of antiregressive work varies from process to process and over the operational life of a long-lived software system.

Software reuse can provide high leverage in the process and help reduce cost and development time. A modeling study of fourth-generation languages revealed that a higher language level reduces the effort in design, code, and approval phases, but it does not reduce effort in the requirements phase.

Using COTS to develop products is a significant change in the way software systems are created, and CBSs continue to be more pervasive. COTS can help lower development costs and time. Using government/legacy off-the-shelf components is quite similar to COTS, but the use of open-source software presents some unique challenges.

With COTS there is no one-size-fits-all model. Assessment activities, tailoring, and glue code development are not necessarily sequential. A COTS glue code model indicated that glue code development has to start at the end of the application development, and the integration process has to start at the beginning of the glue code development.

Software architecting is a success-critical aspect of development that is iterative. An architecting model showed that initial completion rates for requirements and architecture development activities significantly impact the number of approved items. The model also describes a declining curve for staffing analysts and linear growth for architecting and design personnel.

Quality means different things to different people. Generally, stakeholders are concerned about failures that occur during usage or other operational difficulties. Thus, defect analysis is a primary strategy for facilitating process improvement. Defect categorization helps identify where work must be done and to predict future defects. Using separate flow chains to represent different categories of defects is a good way to model defect types or severities with system dynamics.

Project requirements may undergo much change throughout a life cycle. Requirements volatility can be attributed to factors such as user interface evolution, change of mission, technology updates, and COTS volatility. The SPMS model showed the effects of requirements volatility on critical project factors such as cost and schedule. The simulator can also be used to evaluate potential courses of action to address the risk of requirements volatility.

Software process improvement is a major thrust for organizations, involving a lot of investments. A comprehensive process improvement model included subsystems for the life cycle, people, and key process areas to model the timing of improvements to the subsystems. A primary insight of the model centered on people qualities: the attitudinal mix and the pro/con ratio seriously affect the overall potential benefit realized by SPI effort. People with good, proactive attitudes towards process improvement help instill the same behavior in others. Organizations including Xerox have used the model to quantify the duration of process improvement and the value of improving software maturity, using lessons learned by previous organizations that have improved.

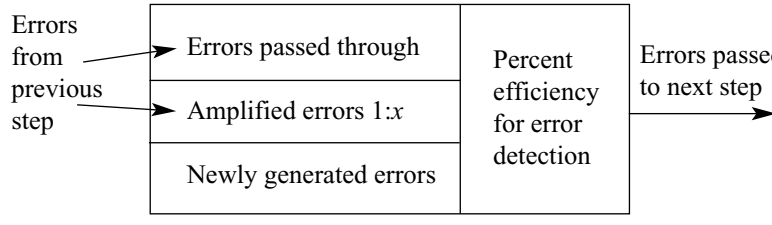
5.14 EXERCISES

These application exercises are potentially advanced and extensive projects.

5.1. Choose a CMMI key process area and develop a model of the covered process activities. Enable your model to evaluate the effectiveness of different process strategies within that area. Higher level process areas based on

process feedback, such as causal analysis and resolution, organizational innovation and deployment, and organizational process performance, are good candidates for automation of process evaluation.

5.2. Develop a defect propagation model based on the diagram below to represent the error flows in a process step. Each step receives errors from a previous phase, some of which are amplified, and new errors are generated before an imperfect detection filter allows errors to be passed on to the next step. See [Pressman 2000] for some worked out examples using a waterfall process. Adapt this kernel to the life-cycle process for your own environment. Alternatively, enhance it to be more granular for different types of error detection.



5.3. Compare and contrast some defect propagation models. Try to use empirical data available from your own environment or the public domain to validate the models.

5.4. a. Develop a defect introduction and removal model calibrated to empirical data for your environment.

b. Develop and validate defect models to account for different severities, types, priorities, or other classification of defects based on empirical data.

5.5. Integrate the provided Madachy and Tvedt inspection models to create a model of both high-level and detailed-level effects.

5.6. Investigate other types of reviews or defect-finding activity by refining the Madachy inspection model. You may also choose other proposed future directions for the model from this chapter or from [Madachy 1996b].

5.7. There are numerous defect-finding and removal methods (e.g., informal peer reviews, formal inspections, pair programming, testing, automated checking, etc.). Many of them overlap in terms of being able to find the same types of defects; hence, there are some inefficiencies when combining techniques. Develop a model that helps determine an optimal order, or sequence, of defect-finding activities.

5.8. Evaluate cost/quality trade-offs based on the COQUALMO extension of COCOMO with a dynamic model. Like system dynamics, it uses a tank-and-pipe analogy to defect introduction and removal. Defects come in through various pipes (e.g., during requirements or design), and linger in the system in "tanks" until they are eliminated through a defect-removal pipe

(e.g., peer reviews or testing). See [Chulani, Boehm 1999], which describes the extension that models defect introduction and elimination rates based on cost driver ratings.

5.9. An iterative version of a Rayleigh staffing curve model using arrays is provided in *Rayleigh iterative.itm* (see the list in Appendix C and Chapter 6 for more details). It enables one to create a project with multiple iterations using offset relationships to set the time phasing between iterations. Adapt and use the model for iterative planning or analysis. For example, the Rayleigh curve can be replaced with formulations for other types of processes and staffing models. The generalized Rayleigh curve generator in Chapter 3 could be integrated into the array structure. The iterative model can alternatively be reformulated into an incremental process model. Also see [Fakharzadeh, Mehta 1999] for an example of modeling process iterations with arrays.

5.10. There has been a large gap observed in COTS-related project effort statistics. Projects have either reported less than 2% or over 10% COTS-related effort, but never between 2% and 10% (see [Boehm et al. 2003]). Why might this be true? What factors might "tip the scale" one way or another on a project? Analyze this phenomenon with a simulation model.

5.11. Create a CBS process model whereby the activities for assessment, tailoring, and glue code development can be interchanged in different sequences. See [Port, Yang 2004] for a study of activity sequences. Experiment with different order sequences. You may also want to refer to [Kim, Baik 1999] for some ideas on modeling COTS-related activity sequences. Draw some conclusions from the model regarding the appropriate conditions for certain sequences. For simplification, you do not have to model all the different sequences that have been observed.

5.12. Investigate COTS lifespan issues suggested by the COTS Lifespan Model (COTS-LIMO) with a dynamic model. It should provide a framework for generating the cost of maintenance curves. A user would perform "what if" analyses by supplying specific values for the parameters in the model and generating sets of contour lines on a case-by-case basis. Controlled experimentation should produce families of breakeven curves for different COTS architectures, and the model should allow one to decide when to retire a CBS.

5.13. Develop some guidelines regarding optimal COTS refresh rates for a specific type of system. Model different dynamic scenarios for COTS product updates and consider overall costs, schedule, and risk.

5.14. Choose another important research issue for CBS that simulation can address and conduct a research project. Read ahead on the COTS section in Chapter 7.

5.15. Investigate the dynamic effects of software development tools across the life cycle. Develop a model so the user can choose different classes of tools and corresponding investment levels for them.

5.16. The PSP method has an important biofeedback-type process for estimation and defect prevention. Model the feedback inherent in the PSP.

5.17. Adapt the Burke process-improvement model for a specific organization. The Xerox adaptation example showed how it was interpreted for a CMM-based process improvement initiative. If your chosen organization is undertaking a Six Sigma initiative, is the model still valid? How might it be adapted for Six Sigma, or should it be superseded by a brand new model?

5.18. Once it is lost, quality is expensive to regain. Model the impact of restarting a process improvement program. Some factors might include a loss of trust or credibility in the organization. Make suggestions about keeping resources optimally allocated for continual process improvement.