A Decision Model for Software Maintenance

Article *in* Information Systems Research · December 2004

DOI: 10.1287/isre.1040.0037 · Source: DBLP

3 authors, including:

M. S. Krishnan
University of Michigan
95 PUBLICATIONS **9,353** CITATIONS

Information Systems Research
Vol. 15, No. 4, December 2004, pp. 396–412
ISSN 1047-7047 | EISSN 1526-5536 | 04 | 1505 | 0396

informs

DOI 10.1287/isre.1040.0037

A Decision Model for Software Maintenance

M. S. Krishnan
Stephen M. Ross School of Business, University of Michigan, 701 Tappan Street, Ann Arbor, Michigan 48109-1234,
mskrish@umich.edu

Tridas Mukhopadhyay, Charles H. Kriebel
Tepper School of Business, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
{tridas@cmu.edu, ck04@andrew.cmu.edu}

In this paper we address the problem of increasing software maintenance costs in a custom software development environment, and develop a stochastic decision model for the maintenance of information systems. Based on this modeling framework, we derive an optimal decision rule for software systems maintenance, and present sensitivity analysis of the optimal policy. We illustrate an application of this model to a large telecommunications switching software system, and present sensitivity analysis of the optimal state for major upgrade derived from our model. Our modeling framework also allows for computing the expected time to perform major upgrade to software systems.

Key words: software maintenance; system replacement; stochastic model; legacy systems
History: Sandra Slaughter, Associate Editor. This paper was received on July 21, 1999, and was with the authors 21.5 months for 5 revisions.

1. Introduction

As computers play an increasingly large role in our society and in our daily lives, it becomes critical for software developers to be able to create and maintain effective software on time, at a minimum cost. Often, software maintenance accounts for 50%–80% of system life-cycle costs for legacy systems (Banker and Slaughter 1997, Banker et al. 1991, Schneidewind 1987). Maintenance activities in software systems are broadly characterized as a sequence of corrective, adaptive, and perfective actions (Swanson 1976). Once a software system has been released to users, initial maintenance efforts may involve corrective work. In the course of time, user requests for various system enhancements may dominate software maintenance.

The adaptive and perfective maintenance of the system may arise due to the changes in the organization's business processes and user needs. The new enhancements made to the system may need further corrective actions. Intermix of these three types of maintenance requests leads to modifications in the existing source code and addition of new source code. Such modifications and additions often tend to increase the complexity of the maintenance task over time for various reasons. First, system code may lose structure due to frequent changes, over time, in various modules of the system by different software engineers. Second, the complexity of the system may increase due to the new changes and potential inconsistencies introduced by the developers, sometimes even without their knowledge (Banker et al. 1993, Gode et al. 1990, Lehman and Belady 1985). Although the significance of understanding the software maintenance process has been emphasized (Lehman and Belady 1985, Swanson and Beath 1989), few decision frameworks have been presented in the research literature to address this problem.

The challenge of controlling software maintenance costs goes beyond the issue of reducing costs by avoiding errors or detecting errors in the early part of the life cycle. Software managers need to adopt their maintenance strategies such that the cost of changes or enhancements to the system is minimized. One reason for the large proportion of software budget attributed to the maintenance activity is software managers' short-term views. Software managers are often faced with budget constraints and schedule pressure in fulfilling maintenance requests from users, and hence focus on incremental maintenance to the system. However, in order to reduce long-term

maintenance costs, it may be economically beneficial to rework the entire software system through major improvements in designs or use of efficient technology (Lehman and Belady 1985, Chan et al. 1996). Although a major rework or replacement of the system may require significant investment, such a rework will also improve consistency in the system, and increase familiarity of the code to the developers (Swanson and Beath 1989).

In this paper, we develop a decision model for the maintenance of large software systems. This model is based on a infinite-horizon, stochastic-discrete, time-dynamic optimization. We derive an optimal policy for initiating major upgrades to the system and relate the propositions derived from this policy to experiences reported in the software literature. We also illustrate our modeling framework through a case study using data collected from the software repository of a leading telecommunications (telecom) switching vendor. The organization of the rest of the paper is as follows. In §2, we discuss the research issues, and in §3 we present the conceptual modeling framework. In §4, we discuss various propositions relating to the effect of software environment on major upgrade decisions in legacy software systems. In §5, we illustrate model application through a case study of a large software maintenance project in the telecom software industry. In §6, we discuss the contributions and managerial implications of our model. We provide conclusions with directions for future work in §7.

2. Research Issues

Scholars have proposed models to predict software reliability or software defects and methods to design software to improve maintainability and reduce errors (Linger 1993, Kung et al. 1994). Empirical studies have emphasized the significance of certain factors, such as size and complexity in determining software maintenance costs (Kemerer 1987, Banker et al. 1991, Banker and Slaughter 1997). It has been recognized that with changes in the usage environment of the software and evolving application domains, enhancement requests from end users in software systems will continue to surface as long as the system is being used. Because software evolves over time as changes are made to it, the complexity of the system and inherent difficulties related to software maintenance tend to increase unless preventive measures are taken to improve the state of the system. For example, preventive measures such as improving and reworking the design, updating documents, and establishing change control to software files may enhance maintainability of the software code. Usually, design improvements and other significant changes to the system are made through a major rewrite or replacement of the entire software system.

In practice, the decision to perform a major rewrite to the software is undertaken when the maintenance has become too expensive, software reliability low, change responsiveness sluggish, system performance not acceptable, or system functionality outdated. However, to minimize software life-cycle maintenance costs, major system upgrades should be timed carefully, taking cost trade-off into account. If a major upgrade is scheduled too early, its cost may outweigh the total costs of minor upgrades to the system. If it is begun too late, however, expected benefits from a major upgrade may be reduced substantially.

A stream of research literature in manufacturing and operations management has addressed the problem of controlling the life-cycle maintenance cost of capital intensive machines used in the manufacturing process. Several models have been proposed to understand the trade-off in hardware or manufacturing machine maintenance environment (Nurani 1995, Chand et al. 1993, Sethi and Chand 1979, Rosenfield 1976, Ross 1971). However, there are some inherent differences between hardware machine replacement and rework on software products. For hardware, major changes to a product are generally achieved by redesign, retooling, and constructing or buying a new instance of the hardware. In software, the code and design are altered and extended to implement a sequence of improvements and adaptations to an evolving application domain and user environment. The new features and functionality in software are often recognized only after users start to use the system. This phenomenon is one of the reasons for the increase in the cost of software rework over time. On the contrary, hardware or manufacturing machine replacement models proposed earlier assume the cost of replacement to be either constant (Sethi and Chand 1979, Rosenfield 1976) or decreasing over time as

a result of learning effects on the machine vendor's manufacturing process (Chand et al. 1993).

An analytical model for economics of software replacement was first presented using a deterministic framework (Barua and Mukhopadhyay 1989, Gode et al. 1990). Chan et al. (1996) extend this approach to include nonzero time for a major upgrade to the system. Although the proposed models provide a good framework to understand the trade-off in software maintenance decisions, they also are applicable to a limited software maintenance environment in practice. For example, these models fix the time horizon of maintenance decisions and require specification of a fixed formulation of maintenance cost functions for the entire time horizon. In addition, these models are static, and do not allow for probabilistic information on the nature of maintenance requests from users. However, in practice, the cost of further maintenance or rewriting of the software depends on the incremental maintenance work done on the system after its release. Hence, in this paper, we present a forward-looking decision framework for software maintenance using a stochastic model to analyze the trade-offs in software maintenance decisions.

3. Modeling Framework

We consider a large software system in which at least some parts have been concurrently implemented. Such software systems often undergo a continuous process of maintenance and evolution. A sample of such systems can be found in many software domains. For example, in the application software domain, custom-built large and aging enterprise application systems with multiple modules fall under this category. As more enhancements are performed, these systems deteriorate and become more expensive to maintain. Such systems are often termed *legacy software systems* (Keith 1995). In this paper, we model the maintenance challenges encountered in such legacy systems. In our model, we define the state of the software system based on the sum of new and modified code in the system. As more additions and modifications are performed on the system, the complexity of maintenance further increases, and the state of the system deteriorates.

We represent the state of the software system as a member in a finite set $S = \{s_j \mid j = 1, 2, 3, 4, \ldots, Z\}$ of integers between 0 and Z. This integer value assigned to the state of the system represents the degree of maintenance work performed in the system since the release of a new software system to the users. At any time t, if the state of the system $s_t = k$, then it indicates that the sum of new and modified code in the system at time t is $(k * 10)\%$ of the initial system size.[1] For example, if the state of the system is 5 with the initial system size 150 KLOC, then the sum of new additions and modifications to the code is 75 KLOC (50% of the initial system size). If the same code of the system gets modified during maintenance in two periods, we count that twice in determining the state of the system. This is due to the fact that the software code loses its structure the more times it is modified for maintenance work.

We assume that the system is initially delivered to the users at State 0 (i.e., with no new additions or modifications in the code due to maintenance). At the beginning of each review period (which can be a planning cycle for software maintenance) the *dm* (decision maker) reviews the state of the system and decides on the maintenance strategy for that period based on the expected current period costs and future costs. Note that we have not explicitly included deletion of code as a part of maintenance in our model for several reasons. First, it is often the case that large segments of software code are not physically deleted during incremental maintenance, and if they are deleted at all, such large segments are only commented out. Even if deleted, this "dead code" adds to the complexity in comprehension. (The case of substantial deletion of code is addressed as part of partial rewrite option to the system addressed later in the paper.) Second, note that minor deletions of few lines of code during incremental maintenance, in a minor upgrade, will not alter the state of the system as per our definition.

We assume that maintenance requests from the users arrive randomly such that the state of the system deteriorates as a stochastically monotone Markov process with a single-step state transition probability matrix P. The deterioration of the system as a Markov

[1] Note that the number 10 is chosen for illustrative purposes. The structure of the model and results will hold for any positive number other than 10. The lower this number, the higher will be the granularity in defining the system state.

process means that P is a $(Z+1)\times(Z+1)$ matrix with elements p_{ij} representing the probability that the state of the system will be j in the beginning of the next review period, given that the state of the system is i in the beginning of the current review period. From our definition of the state of the system, it is clear that the matrix P is upper triangular. That is, more maintenance to the software can only increase the modified and new code added to the software. Hence we have $p_{ij}=0$ for $j<i$.

Although software is obviously different from hardware, we believe that it is still valid to use the Markovian assumption. Note that the basic Markovian assumption is the memory less property of a Markov chain. In other words, the assumption states that the transition probability of moving to a next state depends only on the current state. In the context of our model, because the system state is defined in terms of new and modified lines of code, this state definition reflects the functionality of the software system at that point and includes past changes. Hence, in our model, the next state of the system depends on the amount of change that needs to be implemented in the product, but the required change will certainly depend on only the current state of the system, i.e., all the changes that are in the system now. For a stochastically monotone Markov process, it is also true that the single-step, transition-probability matrix is an IFR (increasing failure rate) matrix (Derman 1963). If P is an IFR matrix, then we have the following property:

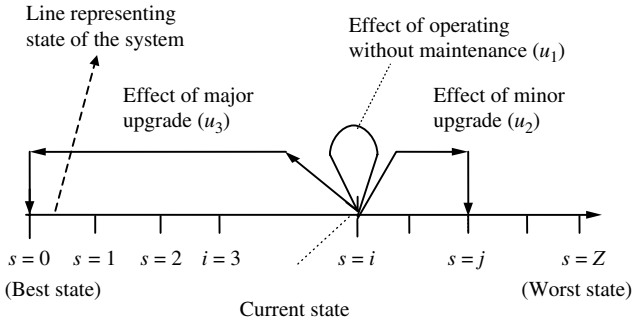
$$\sum_{j=k}^{Z} p_{ij} \text{ is increasing in } i \text{ for all } k\in\{0,1,2,3,4,\ldots,Z\}. \tag{1}$$

In our modeling framework, this condition states that chances of further deterioration in the system are higher for a worse initial state of the system. That is, if i_1 and i_2 are two states such that $i_2>i_1$ (which means i_2 is a worse state), then

$$\sum_{j=k}^{Z} p_{i1j} \le \sum_{j=k}^{Z} p_{i2j} \quad \text{for all } k\in\{i_2, i_2+1,\ldots,Z\} \text{ when } i_2>i_1. \tag{2}$$

In our definition of the state of software system, the above condition is true because, as more new code is added and more modifications to the code are performed, the entropy of the system increases, and consequently the probability of the system deterioration further increases. It has been observed that more changes to software modules will generate further maintenance on the system. The single-period dynamics of the model is depicted in Figure 1. At the beginning of any review period t, let the state of the system be i.

Figure 1 Single-Period State Transition Under Different Maintenance Options



We assume that the *dm* has three options for maintenance in that review period. The first alternative u_1 is to just keep the system operational—that is, leave the system without any maintenance. This option may be observed in practice due to, e.g., budget constraints for that period, understaffing, or the manager's belief about future maintenance requests.

The second alternative u_2 is to order a minor upgrade to the system by fulfilling the users' maintenance requests in that review period. This minor upgrade can also be understood in terms of patch release for the software system. A minor upgrade will involve incremental addition of functionality in the system and few design changes. Thus, a minor upgrade will lead the software to a worse state due to an increase in modified and new code in the system as per our definition.

The third alternative u_3 is to perform a major upgrade involving the rewrite or replacement of the system, thus transferring the system to State 0.

The effort or cost for the three maintenance alternatives u_1 (operational), u_2 (minor upgrade), and u_3 (major upgrade) in each time period would depend on the state of the system in the respective period. Let $\mathrm{OC}(i,j)$ be the cost of operating the system in a review period when the state of the system is i.

Because no maintenance is done when the decision is to keep the system operational, OC(i, j) also includes the penalty cost as a function of the difference between i and j. Similarly, let MC(i, j) be the cost of a minor upgrade to the system when the state of the system is i, and let j be the state of the system in the next period. The difference between i and j indicates, as a percentage of initial system size, the new and modified code from the maintenance requests that arrived in that review period. We denote the cost of a major upgrade to the system when the state of the system is i by RC(i). We next state our assumptions on these functions and provide arguments in support of the same in the context of software systems.

Assumption 1. RC(i) *is increasing in* i *and bounded from above.*

Because the state of the software system deteriorates over time, in our definition, more maintenance work would have been done on the software system. Thus with increase in i, the size of modified and new code in the system increases. After sufficient maintenance work, the system may lose its structure and developers' familiarity with the code may be lower; this leads to an increase in RC(i).

Assumption 2. MC(i, j) *and* OC(i, j) *are bounded and increasing in* i *and* j.

Because more maintenance is done to the system, MC(i, j) and OC(i, j) will tend to increase in i. Because the difference between i and j represents the size of new and modified code from the maintenance requests, it is clear that MC(i, j) and OC(i, j) will increase with increase in j.

Because Z is the worst possible state in our model, these costs are bounded from above. It may be noted that all costs are positive and bounded from below. It is also true in the context of software systems that RC(i) will have a fixed component independent of the state of the system and a variable component depending on the state of the system. The fixed component part captures the costs involved in reinstalling the system, training users, and setting up configuration libraries. Hence, initially for few states after the system is released to the users, RC(i) would be much greater than both MC(i, j) and OC(i, j) even for large values of j. However, as the state of the system deteriorates, RC(i) will not increase as fast for various reasons such as scale economies in rewriting the software (Banker and Slaughter 1997). Hence

Assumption 3. RC(i)−MC(i, j), *and* RC(i)−OC(i, j) *are nonincreasing in* i.

At the beginning of each review period, the *dm* will choose one of the control options u_1, u_2, or u_3. The single-period cost for each of the control options when the state of the system is i are given as follows:

$$\mathrm{SC}(i, u_1) = \sum_{j=i}^{Z} p_{ij}\mathrm{OC}(i, j) \tag{3}$$

$$\mathrm{SC}(i, u_2) = \sum_{j=i}^{Z} p_{ij}\mathrm{MC}(i, j) \tag{4}$$

$$\mathrm{SC}(i, u_3) = \mathrm{RC}(i). \tag{5}$$

Equations (3) and (4) specify the single-period expected cost of minor update and operational choice. Note that because the decision option for a review period is chosen at the beginning of the period, minor update and operational costs are expected costs based on the size of maintenance or enhancement requests in that period. It is assumed that a major update to the software will be completed in one review period. That is, if the major update option u_3 is chosen at the beginning of a review period, then a new system will be ready in State 0 at the beginning of the next period. This assumption can be relaxed to incorporate longer horizons for major updates in the model. The objective of the *dm* is to choose the control options in the beginning of each review period in such a manner that the long-run discounted cost of running the system is minimized. If $0 < \delta < 1$ is the one-period discount factor capturing the present value of future costs, then for any policy π of maintenance decisions adopted by the *dm* the total expected discounted cost is given by

$$L_\pi(i) = \mathrm{E}_\pi\left\{\sum_{n=1}^{\infty}(\delta^n \mathrm{SC}(s_n, u_n)/s_0 = i)\right\},$$

where s_n is the state of the system at the beginning of nth review period and i is the initial state of the system. E_π is the conditional expectation given that the policy π is employed. The *dm*'s objective is to find

an optimal policy π^* that minimizes $L_\pi(i)$, where i is the initial state of the system.

At the beginning of each review period, the *dm* will choose from one of the three control options u_1, u_2, or u_3. The decision model in our setting fits within the framework of Markov decision process with finite state and control space. By the theory of Markov decision process, there exists a stationary randomized optimal policy (Ross 1971). It can be shown there is an optimal discounted cost function for this process starting from any given state i. We denote this function by $L(i)$. The symbols and notation used in our model are defined in Table 1.

Lemma 1. *The functions* $\mathrm{SC}(i, u_1)$ *and* $\mathrm{SC}(i, u_2)$ *are increasing in* i.

Proof. See the appendix.

Theorem 1. *The optimal infinite horizon discounted cost function* $L(i)$*, mapping the state space of the system to the set of real numbers exists and is increasing in* i*, where* i *is the initial state of the system.*

Proof. See the appendix.

Table 1 List of Symbols and Notation Used in the Model

Symbol	Description
0	The best state of the software system
Z	The worst state of the software system
i, j	States of the software system
dm	Decision maker
P	Single-step state transition probability matrix with elements p_{ij}
$\mathrm{RC}(i)$	Major upgrade cost of the system when the state of the system is i
$\mathrm{MC}(i, j)$	Minor upgrade cost when the state of the system is i and the system deteriorates to state j in the next stage
$\mathrm{OC}(i, j)$	Cost of keeping the system operational in the current state i and the penalty cost for not fulfilling the maintenance requests in the current period; $j - i$ represents volume of maintenance requests received.
u_1, u_2, u_3	Control options for the *dm*
$\mathrm{SC}(i, u_1)$	Single-stage cost when state of the system is i and control action taken is u_1
δ	Single period discount factor to capture present value of future costs
$L_\pi(i)$	Total discounted cost function starting from initial state i, when policy π is adopted, i.e., a certain sequence of decision actions π is chosen
$L(i)$	Total discounted cost function starting from initial state i, when optimal policy π^* is adopted
T_i	Expected time for a major upgrade in number of review periods when the initial state of the system is i

The one-step optimal decision rule to minimize the discounted infinite horizon cost is given by

$$L(i) = \min_{u_1, u_2, u_3} \left\{ \mathrm{SC}(i, u_1) + \delta L(i), \mathrm{SC}(i, u_2) + \delta \sum_{j=0}^{Z} p_{ij} L(j), \mathrm{SC}(i, u_3) + \delta L(0) \right\}. \quad (6)$$

That is, out of the control options u_1, u_2, and u_3 the one that minimizes the costs in Equation (6) is chosen. Note that solving this dynamic program gives a complete optimal policy with the decision options based on current cost and expected future cost at every review period. We next show that there exists a state of the system i^* such that it is optimal to do a major upgrade for any state of the system greater than i^*.

Theorem 2. *In the life cycle of software systems, there exists a state* i^* *beyond which it is always optimal to do a major upgrade to the system.*

Proof. See the appendix.

The threshold state value i^* can be interpreted in the following manner. For example, let us assume that the initial size of the software system delivered to the users is 150 KLOC and the value of i^* is 6 for a specific set of cost parameters. Thus it is optimal to provide a major upgrade of the software system once the sum of the new and modified code from postrelease maintenance exceeds 90 KLOC (i.e., 60% of initial system size as defined in §3).

Corollary 1. *The optimal major upgrade state of the system* i^* *cannot decrease* (*cannot increase*) *if the major upgrade cost function* $\mathrm{RC}(i)$ *is replaced with another function* $\mathrm{RC}^d(i)$ *such that* $\mathrm{RC}^d(i)$ *is greater* (*less*) *than* $\mathrm{RC}(i)$ *for all* i*, and satisfies all the assumptions on* $\mathrm{RC}(i)$.

Proof. See the appendix.

Note that the maintenance decision for the system before the state i^* may alternate between minor upgrade and operational decision options depending on the specific cost parameters and functional forms of $\mathrm{MC}(i, j)$ and $\mathrm{OC}(i, j)$, respectively. It is easy to see that the structure of the optimal policy might be states $i^* \geq i_n^* \geq i_{n-1}^* \geq i_{n-2}^* \cdots \geq i_1^* \geq 0$ such that if the optimal action for $i = 0$ is u_1, then the optimal action remains u_1 for all i such $i_1^* \geq i \geq 0$. This is followed by u_2 as the optimal action for all i such that $i_2^* \geq i \geq i_1^*$.

The number of times the optimal control changes between u_1 and u_2 before the threshold state i^* cannot be determined for generalized increasing functional forms of the costs in our model. Also note that the optimal policy derived from our dynamic programming model is stationary, i.e., it depends only on the current state of the system and not on the specific review period. We next discuss the optimal state for the first major upgrade after releasing the software system to the users at State 0 (that is, when a new system has been released to the users and no additions or modifications have been performed).

4. Software Maintenance Implications

In this section we examine various propositions relating to the effect of software environment on major upgrade decisions in a legacy software system.

4.1. System Major Upgrade: Time and Planning Horizon

In practice, it is observed that some legacy software systems are continuously maintained with minor upgrades, and no major effort for restructuring the system is undertaken. Software organizations continue to maintain such systems for many years. Our model aids in explaining this phenomenon from a cost perspective based on stochastic maintenance requests.

Corollary 2. *Consider a finite time horizon of first n review periods after the release of the software system. Let s_n be the state of the system in the beginning of nth review period. If s_n is less than i^*, then it is not optimal to initiate a major upgrade of the system within the time horizon of n review periods.*

Proof. The proof follows trivially from Theorem 2.

The explanations of this phenomenon of legacy systems being maintained with minor upgrades for a long time are based on several reasons within our framework. First, it may be true that maintenance requests do not arrive in bursts, and are spread over a long time period (Keith 1995). Thus the transition matrix P may be such that the system deteriorates at a slower rate. The second reason is based on the possible increase in major upgrade costs of the system. In a long time horizon, the software organization may lose a significant fraction of the original developers. As a result, the cost of providing a major upgrade for the system may increase (Rugaber and Doddapaneni 1993). Consequently, the optimal state i^* for major upgrade may be much higher. The third reason may be due to the nature of penalty costs for not fulfilling users' maintenance requests. In particular, if the penalty costs are not too high, the control option u_1 of keeping the system operational may be optimal for a large number of system states. Thus, the optimum state for the first major upgrade i^* may be higher. Note that we have so far made an assumption in our model that a major rewrite of the software can be accomplished within a single period. However, this may not be true in the case of large systems. We next show how relaxing this assumption will not affect the overall structure of results in our decision model. The only change will be a shift in the optimal state for major updates i^*.

4.2. Multiperiod Major Upgrade of a System

Let us assume that a major rewrite of a large system can be completed in N review periods instead of one. In our decision framework we can modify the decision option u_3 in the following way to include this change: u_3 can be interpreted as a decision option to keep the current system without any maintenance for N review periods, and work on the rewrite of the new system. In other words, the cost of major upgrade now also includes an additional term, i.e., the penalty cost of keeping the existing system operational without any maintenance. Note that this additional cost will increase with the length of duration for a major rewrite of the software system and does not violate any of our assumptions on various cost functions. Hence the structure of the optimal policy still holds, and because the optimal policy is stationary, the decision option depends only on the state of the system and not on the review period. This new interpretation of decision option u_3 is the same as replacing the major upgrade cost function RC(i) with a new function that is always greater than the original cost function for major upgrade for all state values as stated in Corollary 1. It is clear from that corollary that if the major upgrade cannot be completed within a single review period, then the optimal state i^* for a major upgrade will increase. In addition, this increase in i^* will be higher for longer duration of

major upgrade, i.e., with increase in N. We have also illustrated this point in the case study presented later in the paper. This longer duration for a major upgrade is yet another reason why organizations continue to use software systems that were developed in COBOL over 20 years ago.

4.3. System Major Upgrade and Technology Choice

It is quite common in practice that legacy systems (for example, COBOL programs) often witness a major upgrade or complete replacement using a superior technology such as higher-generation languages and object-oriented methods (Sneed and Nyary 1994). These superior technologies are believed to be more structured and provide better capabilities in terms of the various features and performance that can be offered. In addition, the superior technology promises to be cheaper and flexible, and can potentially enable major upgrades to the system at a reduced cost (Dean et al. 1994, Booch 1993). We next examine the change in the threshold state for major upgrade i^* with a choice of a superior technology.

Let $\mathrm{OC}^n(i, j)$, $\mathrm{MC}^n(i, j)$, and $\mathrm{RC}^n(i)$ represent the cost functions for decision choices u_1, u_2, and u_3, respectively, with a superior technology. On the same lines, let $\mathrm{OC}^o(i, j)$, $\mathrm{MC}^o(i, j)$, and $\mathrm{RC}^o(i)$ represent the same costs with the current technology of the system. As discussed above, we have the following inequalities holding true for the superior technology:

$$\mathrm{RC}^o(i) > \mathrm{RC}^n(i), \quad \mathrm{MC}^o(i) > \mathrm{MC}^n(i), \quad \text{and} \quad \mathrm{OC}^o(i) > \mathrm{OC}^n(i) \quad \text{for all } i. \tag{7}$$

Let P^o and P^n be the transition probability matrices representing the system deterioration with the current technology and the superior technology, respectively. As discussed earlier, the superior technology is more flexible and the system may deteriorate slower because changes can be accommodated in the system without much addition of new or modified code to the system. Hence, it is reasonable to assume that the difference matrix $R = P^o - P^n$ is monotonically increasing left to right in each row. Note that the matrix R is a zero sum matrix, i.e., the sum of each row in this matrix is zero.

Proposition 1. *If the system is changed with a major upgrade using a superior technology, then*

(1) *the optimal discounted cost for using superior technology starting from any state i cannot exceed that for the current technology; and*

(2) *the optimum first major upgrade state i^* using the superior technology is lower than that for the current technology.*

Proof. See the appendix. Note that we assume that the new technology deteriorates slower, hence costs of major and minor upgrades are lower.

4.4. Partial Major Upgrades to the System

The u_3 decision option in our model involves a complete rewrite of the entire software system. However, in practice this may not always be feasible. Software managers may consider a partial rewrite of the system where some parts of the modified code are redesigned. In the context of our definition, such a partial rewrite to the software will improve the state of the system.

Let us assume that *dm* has a fourth maintenance option (u_4) of doing a partial rewrite of the software to bring the current state i back to a predetermined state k $(0 < k < i < Z)$. For example if $k = 3$, and the current state of the system is, say, 5 (i.e., the size of new and modified code is 50% of the original size), the *dm* can provide a partial rewrite to the system by redesigning and rewriting some portions of the modified code, and bring the state of the system to 3. It is important to highlight the difference between a minor upgrade and a partial rewrite in our model. Whereas a minor upgrade is similar to a patch release to the system that either modifies the system code or adds new code, a partial rewrite is a complete rewrite of a major component or subsystem code in order to improve system entropy. It may also include deletion of some code, or "code clean up," where some modules are rewritten. Note that partial rewrite of the software system is relevant only when the current state of the system is higher than k. Let $\mathrm{PRC}(i)$ be the cost of partial rewrite to bring the system back to state k when the current state of the system is i. Note that the cost of partial rewrite to the system increases in i for any state $i > k$. This is true in the case of software systems because the effort needed to bring a system with larger modified and new code to the state k increases with i. Hence, the following assumption:

Assumption 4. PRC(i) *is increasing in* i *for all* $i > k$.

Because the partial rewrite option is not relevant when the state of the system is better than k, we assign a very large number $\text{PRC}(i) = \Delta$ for $i < k$ so that this option is never chosen for any state $i < k$. Let $\text{SC}(i, u_4) = \text{PRC}(i)$ for all i. Note that the difference between RC(i) and PRC(i) will decrease with increase in $i-k$ for $i > k$. For example, let $k = 3$, and the current state of the system $i = 4$, that is, the size of modified and new code added to the system is 40% of the initial system size. Then, PRC(i) is the cost of effort needed to rework only the additional 10% of modified and new code, whereas RC(i) is the cost of effort needed to rewrite all the modified and new code. Hence PRC(i) will be substantially lower than RC(i). However, if the current state is $i = 15$, for example, then the difference between PRC(i) and RC(i) will decrease due to the economies of scale. This leads to our next assumption:

Assumption 5. $\text{RC}(i) - \text{PRC}(i)$ *is decreasing in* i *for* $i > k$.

The one-step decision rule for the decision actions at any period including the partial rewrite option to minimize the system life-cycle maintenance cost is now given by

$$L(i) = \min_{u_1, u_2, u_3, u_4} \Big\{ \text{SC}(i, u_1) + \delta L(i), \quad \text{SC}(i, u_2) + \delta \sum_{j=0}^{Z} p_{ij} L(j), \quad \text{SC}(i, u_3) + \delta L(0), \text{SC}(i, u_4) + \delta L(k) \Big\}. \tag{8}$$

That is, out of the control options u_1, u_2, u_3, or u_4, the one that minimizes the costs in Equation (8) is chosen. Note that, from Assumption 4, the results of Theorem 1 are valid when the control option u_4 is included and the optimal cost function $L(i)$ defined in Equation (8) is increasing in i.

Corollary 3. *In the presence of a partial rewrite option for the dm, there exists a state* g^* *beyond which it is always optimal to do major upgrade to the system.*

Proof. See the appendix.

Note that we have added a fourth option to the decision choice and the optimal state for major upgrade is defined in the proof of Theorem 2 as the first state when the three decreasing difference functions cross zero. Thus, when partial upgrade is considered as a decision option, the optimal state for major upgrade cannot be less than the optimal state without this option. That is, the consideration of partial upgrade as a decision option will tend to increase the optimal state for a major upgrade. In other words, a partial upgrade may be used by software managers to prolong the life of an existing system without adverse implications on long-term maintenance costs.

4.5. System Major Upgrade and Maintenance Environment

In the prior models of software replacement, we have argued that increased productivity of maintenance environment will delay the optimal time for a major upgrade (Chan et al. 1996). These models assume a fixed cost to rewrite the software. Similarly, in our framework, we can assume that the cost of major upgrades, state transition matrix P, and operational costs are the same for the two maintenance environments a and b, and only the cost of minor upgrades $\text{SC}^a(i, u_2)$ and $\text{SC}^b(i, u_2)$ are such that $\text{SC}^a(i, u_2) \geq \text{SC}^b(i, u_2)$ for all i. Then it can be easily shown that the optimal state for major upgrade in environment b cannot be earlier than that in environment a. Note that the above argument stems from the assumptions that the cost of a major upgrade and the system deterioration process are the same. However, if any of these assumptions are dropped, it may not be possible to state in which environment the optimal state for a major upgrade will be lower.

Prior models have also argued counterintuitively that a more volatile environment with a higher maintenance cost will delay a major upgrade (Gode et al. 1990, Chan et al. 1996). This result stems from the assumption that maintenance cost decreases significantly in the new environment after a major upgrade. This assumption is valid especially if the primary challenge of maintenance is due to the complexity of the existing software, and not due to changing user requirements. It is possible to get similar results in our model. However, in practice, when a major software upgrade is undertaken, the software maintenance team applies the knowledge learned about the user environment to improve productivity. Moreover, a major upgrade gives flexibility to reduce the

complexity of the system. Hence, if a system environment generates more maintenance requests, it is important to identify the source and nature of maintenance requests to determine the optimal state for a major upgrade.

4.6. Optimal Policy and Optimal Time for a Major Upgrade

As noted earlier, the dynamic program presented here gives the full policy with the optimal control options for every review period. We have so far discussed the propositions in the context of the optimal state for a major upgrade. However, one point of interest to software managers is the optimal time (in terms of number of review periods) for a major upgrade. Note that, in our framework, deterioration of the state of the system is not known for certain—it is stochastic. In addition, the optimal policy is stationary. That is, the optimal decision option depends only on the state of the system and is independent of the review period. Hence, the number of review periods after which a major upgrade will be undertaken is a random variable. It is possible to compute the mean of this random variable, i.e., the expected number of review periods after which a major upgrade to the system will be optimal.

Let us assume that the single-step optimal decision rule specified in Equation (6) is used to derive periodic optimal maintenance policy for the software system. Let Q be the one-step state transitional probability matrix such that its elements q_{ij} represent the probability of reaching state j at the end of a review period by adopting the optimal policy when the state of the system at the beginning of the review period is i. Note that the matrix Q is different from the matrix P. Q is the state transition matrix when the optimal decision rule is adopted. If i^* is the optimal threshold state as defined in Theorem 2, it is always optimal to do a major upgrade for all states i when $i > i^*$. Let T_i be the expected time for a major upgrade when the initial state of the system is i. Then T_i is given by

$$T_i = \sum_{j \le i^*} q_{ij}(T_j + 1) + \sum_{j > i^*} q_{ij}. \tag{9}$$

In the above equation, T_i is computed based on the probability of the system moving to the next state and the expected time for a major upgrade starting from that state. The first term is the summation for all the states up to i^*, and the second term is the summation for all the states above the threshold state i^*. In the first term, the one period for moving from state i to state j is added to T_j. Note that for all the states above i^*, a major upgrade to the system is immediate. Hence, for those states, the expected time for major upgrade is one in the second term of Equation (9). Given the matrix P and the optimal policy, it is possible to compute T_i for all i. As discussed earlier, software managers are interested in the first major upgrade to the system when the initial state of the system is 0, i.e., T_0. We next illustrate the computation of optimal policy and the state i^* for major upgrade and the time for major upgrade in a case study drawn from a large software system from the telecom industry.

5. Case Study

Our research site is a large software development lab of Lucent Technologies. This lab develops software for a variety of switching systems ranging from large central office telecom switches to small business private switches. Software systems developed for large telecom switches are highly complex, and exceed a million lines of code in size. These software systems exhibit all the characteristics of complex software systems described earlier (Gelman et al. 1992, Jones 2000). The primary customers for these switching systems are large telecom and Internet service providers. These systems are often developed for a custom platform, and incremental maintenance jobs are performed on these systems to meet customer needs and changes in the telecom network environment.

Our interviews with the software managers and developers at the research site revealed that once the switching software is released to the field, new features and bug fixes are delivered periodically to the customers. As a consequence, existing system code is modified, and new code is added to the system. In an attempt to improve the maintenance environment for the software system, the firm had initiated a study to understand the deterioration in the software, and retrieved all the changes made to the system from their code management system archive. In this process, we gathered detailed information on the number

of lines of codes added and modified for each maintenance request.

The analytical model described earlier was then applied to the large software system based on the data collected from the code management library and other inputs from the managers. The initial system size was 100 thousand lines of code. Based on inputs from managers and data on the maintenance history of prior systems in the same domain, we determined that there are 14 states of the system, and defined the set of states as $S = \{0, 1, 2, 3, \ldots, 13\}$ in our model. We noted from the maintenance data of prior systems that major upgrades usually are performed before 130% of the original code is changed or rewritten. That is, once the sum of the new and modified code in the system reaches $13 * 10 = 130\%$ of the initial system size (100 KLOC for this system), the system is often replaced with a major rewrite. We examined the maintenance decision choices faced by the software manager in every period, and applied our maintenance model to this scenario. The review period for maintenance decisions in this environment was six months. At the beginning of the first period, after the system release, the state of the system is zero because no modifications or additions have been performed on the code. As noted earlier, the manager has four decision options at the beginning of each review period.

As noted earlier, the state transition probabilities are derived based on the data collected on maintenance requests from similar telecom software environments. Based on data from maintenance history of prior systems, we estimated the distribution of maintenance requests for every six months as a normal distribution with a mean of 10 KLOC and a standard deviation of 3 KLOC. Because the system deteriorates over time and the probability of getting more maintenance requests increases as the state of the system deteriorates, we expect the probability of the volume of maintenance requests to change with the state of the system. Hence we construct the state transition matrix P in the following manner: For a given state i, let Di be a normal random variable with mean $\mu = (10 + 5 * i)$ and standard deviation $\sigma = 3$. Note that the mean of this distribution drifts with the state of the system. Let S_n and S_{n+1} represent the state of the system in the review periods n and $n+1$ respectively. The transition probabilities are defined as in Equation (10).

$$p_{ij} = \Pr(S_{n+1} = j \mid S_n = i) = \begin{cases} 0 & \text{for } j < i \\ \Pr(j-i) * 10 < Di < (j-i+1) * 10 & \\ & \text{for } j = i, i+1, \ldots, Z-1 \\ 1 - \sum_{j=i}^{Z-1} p_{ij} & \text{for } j = Z. \end{cases} \quad (10)$$

The matrix created in this way satisfies the assumptions of increasing failure rate specified in Equation (1). The cost specifications for the various decision options for a given state of the system i are as follows:

(1) Cost of not doing any upgrades to the system and leaving the system operational $\mathrm{OC}(i, j) = \alpha 10 + \alpha 11 * i^a + \alpha 12 * ((j - i) * 10)^{g1}$, where $\alpha 10$, $\alpha 11$, $\alpha 12$, and $g1$ are cost parameters. $\alpha 10$ is the fixed cost, $\alpha 11$ is the cost that depends on the current state of the system, and $\alpha 12$ is the one-time penalty for not fulfilling the maintenance requests. Note that the penalty cost is proportional to the size of the maintenance requests received in that review period.

(2) Minor upgrade cost incurred when the current state of the system is i and the next state of the system is j is given by $\mathrm{MC}(i, j) = \alpha 20 + \alpha 21 * i^b + \alpha 22 * ((j - i) * 10)^{g2}$, where $\alpha 20$, $\alpha 21$, $\alpha 22$, and $g2$ are cost parameters. Similar to earlier specification, $\alpha 20$ captures the fixed cost incurred in all the minor upgrades and $\alpha 21$ captures the part of the minor upgrade cost that changes with the state of the system. As explained earlier, the cost of doing a minor upgrade will increase as the state of the system worsens. The third expression in the equation is the part of the cost that is driven by the number of new and modified source lines of code in the minor upgrade. Based on our state definition, $(j - i) * 10$ represents the percentage of the initial system size modified or changed, and the initial system size is 100K. Note that our cost formulations are such that the minor upgrade cost changes with the current state of the system and allows for learning effect over time through cost parameters b and $g2$.

(3) Cost of a major upgrade from state i is given by $\mathrm{RC}(i) = \alpha 30 + \alpha 31 * (i * 10)^{g3}$, where $\alpha 30$, $\alpha 31$, and $g3$

are cost parameters. Here $\alpha 30$ is the fixed cost of a major upgrade whereas the second part of the cost depends on the size of modified and new code when the state of the system is i. We have also included the partial rewrite option in this case study. For simplicity, we have assumed the partial rewrite parameter discussed earlier in §4.5, as $k = 5$. That is, for all states of the system greater than five, the software manager has an option for partial rewrite of the system by redesigning and rewriting some portions of the code and bringing the state of the system to $k = 5$. That is, the size of new and modified code is 50% of the initial system size.

In our case study illustration, the cost parameters are $\alpha 10 = 40$, $\alpha 11 = 12$, $\alpha 12 = 5$, $\alpha 20 = 10$, $\alpha 21 = 10$, $\alpha 22 = 5$, $\alpha 30 = 200$, $\alpha 31 = 0.75$. The size exponent parameters in the cost functions are $g1 = 0.25$, $g2 = 0.95$, $g3 = 0.75$, the constants $a = 2.5$, $b = 1.5$, and the discount factor $\delta = 0.95$. Note that these cost functions are in line with the traditional cost functional specifications in prior software research, and all our assumptions are valid for these specifications (Banker et al. 1991, Banker and Slaughter 1997). These cost parameters were also confirmed with the software managers at our research site to check if these specifications are in line with the maintenance efforts incurred at that site. The transition probability was derived from the details on the history of maintenance in the code management library of this software lab. As discussed earlier, in our definition of the system state the transition probability can be obtained from the information on the size of new lines added and existing lines of code changed for each maintenance request. Note that the size exponent factors are less than one, indicating the presence of economies of scale in software maintenance as reported in earlier studies (e.g., Banker and Slaughter 1997). It may be noted that the assumptions stated earlier in our modeling framework hold for these cost functions. We solved the dynamic program in Theorem 1 using the value iteration procedure (Bertsekas 1987). The value iteration procedure starts with a constant optimal cost function $L(i)$ for all states. The dynamic program specified in Theorem 1 is solved to obtain new values for the cost function $L(i)$ for all the states. This procedure is continued iteratively until the cost functions $L(i)$ converge to a stable fixed function.

Based on the cost parameters discussed above, we derived the full optimal policy and optimal decisions for each review period as shown in Table 2. Note that the inclusion of partial rewrite as a decision option can significantly alter the optimal policy. Hence, we report the optimal state for major upgrade with and without the consideration of a partial rewrite option in Table 2. As depicted in Table 2, we find that for telecom switching software with these cost parameters, the state of the system after which it is always optimal to perform a major upgrade is six ($i^* = 6$) when partial rewrite is not considered. That is, once the sum of modified and new code added in the system is equal to or exceeds 60% of the initial size, it is always optimal to perform a major upgrade through system rewrite. The optimal expected total maintenance cost (effort measured in person months) for following this policy is 784 person months. The average time after which a major rewrite option becomes optimal is approximately six review periods, or three

Table 2 Model Results—Normal Business Scenario

	State of the system													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Scenario 1 (Normal change requests)														
No upgrade (1)	44	96	98	140	217	467	802	1,228	1,753	1,379	3,094	3,094	4,838	N/A
Minor upgrade (2)	57	70	92	121	156	194	237	283	333	385	441	498	557	N/A
Partial rewrite (3) $k = 5$	N/A	N/A	N/A	N/A	N/A	168	172	182	194	210	220	234	245	258
Major upgrade (4)	200	204	207	209	211	214	216	218	220	221	223	223	227	228
Optimal action	1	2	2	2	2	3	3	3	3	3	3	4	4	4

Notes. Mean time for major upgrade to be an optimal action with partial rewrite $(i^*) = 11$.
Mean time for major upgrade to be an optimal action without partial rewrite $(i^*) = 6$.

Figure 2 **Impact of Major Replacement Cost on Optimal Replacement State**

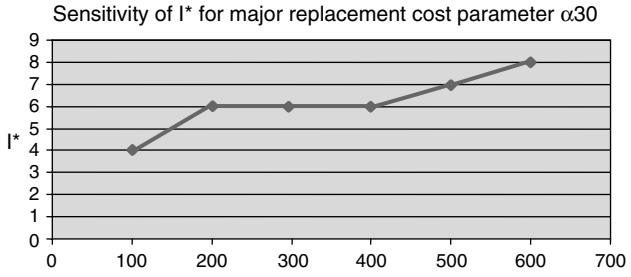
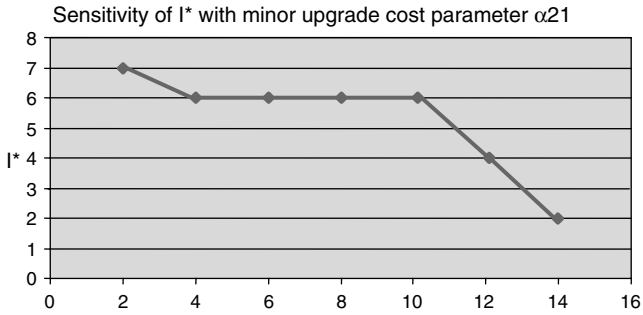


Figure 3 **Impact of Minor Replacement Cost on Optimal Replacement State**



years in this example. Note that the optimal major upgrade state (i^*) may differ with changes in the cost parameters and when a partial rewrite is included. We next discuss the results of the sensitivity analysis of changes in i^* with respect to the parameters of major and minor upgrade costs.

Figure 2 depicts the relationship between optimal major upgrade state (i^*) and the major upgrade cost parameter $\alpha 30$. As stated in §3, it is evident that if all other costs parameters are held the same, with increase in the cost of a major software upgrade RC(i), i^* increases. This scenario is usually seen when the initial designers of the system are no longer in the software organization, and hence the cost of rewriting the modification to the system becomes increasingly difficult. Similarly, Figure 3 depicts the relationship between maintenance cost parameter $\alpha 21$ and i^*. Here we find that with increases in the minor upgrade cost MC(i, j), i^* decreases. Similar sensitivity analysis may be conducted for the various other cost parameters to understand the changes in the optimal major upgrade state i^*.

Note that i^* may also vary with the mean and variance of the stochastic distributions of the maintenance requests for a given systems environment as represented in the transition probability matrix P in Equation (10). We illustrate in Table 3 the optimal policy derived using a different transition probability matrix P^1 that represents a more volatile maintenance environment where the system deteriorates faster due to the higher volume of change requests. As discussed earlier, the optimal state for major upgrade now decreases ($i^* = 4$) and the mean time for a major upgrade also decreases to two years (without considering the partial rewrite). The stochastic model presented here can be used to aid major upgrade decisions in both new software systems and in existing software systems currently in use. In the case of new systems, based on data collected on maintenance

Table 3 **Model Results—Volatile Business Scenario**

	State of the system													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Scenario 2 (More volatile business environment)—more change requests														
No upgrade (1)	479	556	650	569	738	989	1,309	1,784	2,311	2,937	3,654	4,470	5,401	N/A
Minor upgrade (2)	517	575	631	647	705	746	791	843	895	949	1,005	1,062	1,121	N/A
Partial rewrite (3) $k = 5$	N/A	N/A	N/A	N/A	N/A	595	601	617	630	646	648	666	674	678
Major upgrade (4)	637	641	644	647	649	651	653	655	657	659	661	663	664	666
Optimal action	1	1	2	2	4	3	3	3	3	3	3	4	4	4

Notes. Mean time for major upgrade to be an optimal action with partial rewrite (i^*) = 11.
Mean time for major upgrade to be an optimal action without partial rewrite (i^*) = 4.

Table 4 Model Results—Major Upgrade in Two Review Periods

		State of the system													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Scenario 3															
When upgrades	No upgrade (1)	44	96	98	140	217	467	802	1,228	1,753	1,379	3,094	3,094	4,838	N/A
cannot be done	Minor upgrade (2)	57	70	92	121	156	194	237	283	333	385	441	498	557	N/A
in a single	Partial rewrite	N/A	N/A	N/A	N/A	N/A	188	191	214	228	243	260	274	289	298
review period	($k = 5$) (3)														
Major upgrade	Major upgrade (4)	261	264	266	269	271	273	275	277	279	281	283	284	286	288
takes two	Optimal action	1	2	2	2	2	3	3	3	3	3	3	3	4	4
review periods															

Notes. Mean time for major upgrade to be an optimal action with partial rewrite (i^*) = 12.
Mean time for major upgrade to be an optimal action without partial rewrite (i^*) = 7.

requests in prior products, software organizations can estimate the distribution of the maintenance requests and use the stochastic decision model presented here. In the case of software systems already in use, this model can be used by estimating the distribution of maintenance requests in the current maintenance environment, based on existing maintenance data.

As discussed in §4.2, the assumption in our model that a major upgrade is completed in a single review period can be relaxed. We illustrate this in Table 4, by assuming that it takes two review periods for a major upgrade to be implemented and assigning a penalty cost for inability to fulfill the maintenance requests that arrive during this time. All other cost parameters are fixed the same. As shown in Table 4, the optimal state for major upgrade increases ($i^* = 7$) due to the increased time needed for completing a major upgrade.

6. Contributions and Discussion

Information systems in organizations either continually change or become obsolete and less useful. It is recognized that managers of large software systems need to base their planning process on models and facts in order to gain insights about their maintenance process and assess various alternatives quantitatively. In this paper, we presented a general stochastic dynamic framework to compute the optimal state for a major upgrade to the system. Our model presents a new decision framework that is unlike earlier models on the economics of software replacement. First, the optimal time for software replacement in the earlier models was derived, conditional on the length of the planning horizon (Gode et al. 1990, Chan et al. 1996). Note that although the prior models expect the managers to define a time period T for the system horizon, we define a worst state Z. This places different requirements on the data needed for the two types of models. Whereas the prior models anticipate that managers can predict that the system will reach its worst state after a fixed number of years, our model expects managers to predict that it will be difficult to maintain the system after a certain percentage of initial code is new or changed. Often the latter is easier to estimate because it sets a limit to the magnitude of accumulative changes feasible to implement, but it does not require how soon the changes will materialize based on corrective adaptive and perfective changes. These two scenarios will be similar if we can assume that the maintenance requests arrive at a known constant rate over time. But this is seldom true in the real world. It is important to note that our model allows for the stochastic nature of maintenance requests.

As shown in the case study, an estimate of this stochastic distribution can be obtained from the source code management system of similar software systems. This assumption regarding stochasticity in maintenance requests also makes the optimal time to replace the system a random variable; we believe that this approach presents a different perspective on software maintenance for managers to complement the existing static deterministic models. Software managers can create the initial transition probability matrix based on data on maintenance requests from past projects in the same environment. Managers can

also update their probability and reestimate their optimal decisions by running the algorithm again in our forward-looking decision model. Another important distinction between our model and the prior models is that our model presents the full optimal policy, i.e., the set of decision options at each review period based on the actual maintenance requests seen by the software manager. The optimal control policy is updated at each review period, and the full policy is presented with optimal controls and costs associated with each control option for every period. This is distinctly different from the decision solution derived from prior static models where the primary focus is on the optimal time to replace and the decision problem is modeled as a one-time decision. Our model also allows for a partial rewrite of the software system that is not covered in most prior software maintenance models.

We want to emphasize that the two modeling approaches are complementary; it is important for managers to be aware of both approaches because they may be applicable in different maintenance scenarios. We believe that the contributions of such decision frameworks need to be viewed both from the perspective of the results and the relevance of these models for practical use to software managers. In a large software development environment, managers often plan based on the realization of requests for new features or corrective fixes. For example, in the case of a software environment where the uncertainty in maintenance pattern is lower and the cost of maintenance is not significantly affected by any incremental changes, a static model may be appropriate. This situation may be common in large legacy application software in a stable business domain. However, a dynamic model may be appropriate in a volatile software domain, where the uncertainty in maintenance requests is higher due to changes in the business domain and evolution of technology, and the effort required to implement new maintenance changes significantly depends on the volume of existing maintenance.

7. Conclusion

The benefits from our model depend on the accuracy of the estimates of the distribution of maintenance requests and cost parameters. However, in practice, we believe that our stochastic model based on the distribution of the maintenance requests will add richness to the inference from single-period static models proposed for software upgrades in prior work. In addition, with an institutionalization of a maintenance metrics program to collect longitudinal data on maintenance requests, software organizations will derive added insights on the cost implications of their upgrade decisions. It should be noted that the quality of any decision model is limited by the quality of the input data, hence it is important for software development organizations to emphasize a disciplined metrics program on software maintenance in meeting the same.

Our model is a first attempt at deriving the optimal time for a major system upgrade, based on the uncertainties in the user environment and product performance. Incorporating maintenance backlogs can be a natural extension to this model. The option for partial major upgrade to the software in our model is restricted by the assumption by a fixed state k for all the partial major upgrades. However, the choice of k may be included as a control option for the *dm*. The framework presented here could also be extended to model portfolio decisions when a *dm* manages a number of software maintenance projects.

Acknowledgments

The authors are grateful to John Botsford, Karen Bennet, and Larry Votta for their support and valuable suggestions. Financial support for this research was provided through a fellowship from the Software Engineering Institute, Carnegie Mellon University, and from the Mary and Mike Hallman Research Fellowship at the Ross School of Business, University of Michigan.

Appendix

Lemma 1. *The functions* $\mathrm{SC}(i, u_1)$ *and* $\mathrm{SC}(i, u_2)$ *are increasing in* i.

Proof. We know that the state transition probability matrix P is an IFR matrix and satisfies Equation (2). Because $\mathrm{OC}(i, j)$ and $\mathrm{MC}(i, j)$ are increasing functions in i and j from Assumption 2, $\mathrm{SC}(i, u_1)$ and $\mathrm{SC}(i, u_2)$ are increasing in i. Q.E.D.

Theorem 1. *The optimal infinite horizon discounted cost function* $L(i)$, *mapping the state space of the system to the set of real numbers exists and is increasing in* i, *where* i *is the initial state of the system*.

PROOF. Let K^0 be a function that maps the state space S to the set of real numbers. Let K represent an operator on the function K^0 defined as follows: $K^0(i) = 0$ for all i.

Applying the operator K to the function K^0 once, we get the function K^1 defined as

$$K^1(i) = \text{Min}\Big\{\text{SC}(i, u_1) + \delta K^0(i), \text{SC}(i, u_2) + \delta \sum_{j=0}^{Z} p_{ij} K^0(j), \text{SC}(i, u_3) + \delta K^0(0)\Big\}.$$

Similarly, when operator K is applied repeatedly q times, the function K^q is given by

$$K^q(i) = \text{Min}\Big\{\text{SC}(i, u_1) + \delta K^{q-1}(i), \text{SC}(i, u_2) + \delta \sum_{j=0}^{Z} p_{ij} K^{q-1}(j), \text{SC}(i, u_3) + \delta K^{q-1}(0)\Big\}.$$

We now show by mathematical induction that the function K^q is a nondecreasing function in i. For $q = 0$, by definition, $K^0(i) = 0$ is a nondecreasing function in i. Because the minimum of three nondecreasing functions in i is also nondecreasing in i, for $q = 1$ the function $K^1(i)$ is again nondecreasing from Assumption 1 and Lemma 1.

Let us assume for $q = r$, $K^r(i)$ is a nondecreasing function in i. Now for $q = r + 1$

$$K^{r+1}(i) = \text{Min}\Big\{\text{SC}(i, u_1) + \delta K^r(i), \text{SC}(i, u_2) + \delta \sum_{j=0}^{Z} p_{ij} K^r(j), \text{SC}(i, u_3) + \delta K^r(0)\Big\}.$$

Because we have $K^r(i)$ as a nondecreasing function in i, by the same argument as above, $K^{r+1}(i)$ is by itself a nondecreasing function in i. Thus, by induction, we find that $K^q(i)$ is a nondecreasing function in i for all q. Note that by definition $K^q(i)$ is the minimum discounted cost function for a q period problem. However, these K^q functions are bounded from above (because $\text{SC}(i, u_t)$ for $t = 1, 2, 3$ are bounded). Hence, as q tends to a large value, these functions converge to a fixed function as a result of contraction mapping (Bertsekas 1987). We denote this function by $L(i)$.

$$L(i) = \lim_{q \to \infty} K^q(i).$$

Because $K^q(i)$ is a nondecreasing function in i for all q, $L(i)$ is nondecreasing in i. Q.E.D.

THEOREM 2. *In the life cycle of software systems, there exist a state i^* beyond which it is always optimal to do a major upgrade to the system.*

PROOF. We know that the optimal maintenance policy is the decision rule given in Equation (6). We define two functions $D_{13}(i)$ and $D_{23}(i)$ in the following way: $D_{13}(i)$ represents the difference in cost of a major upgrade (u_3) and the choice of keeping the system operational without any maintenance (u_1) when the state of the system is i. Similarly, $D_{23}(i)$ is the difference in the cost of a major upgrade and minor upgrade (u_2) when the state of the system is i. From Equation (6), we get

$$D_{13}(i) = \text{SC}(i, u_3) - \text{SC}(i, u_1) + \delta L(0) - \delta L(i) \quad \text{(A1)}$$

$$D_{23}(i) = \text{SC}(i, u_3) - \text{SC}(i, u_2) + \delta L(0) - \delta \sum_{j=0}^{Z} p_{ij} L(j). \quad \text{(A2)}$$

We first show that $D_{13}(i)$ is decreasing in i. From Assumption 3, we know that $\text{SC}(i, u_3) - \text{OC}(i, j)$ is decreasing in i. Because $\text{OC}(i, j)$ is increasing in i for all j, we have $\text{SC}(i, u_3) - \text{SC}(i, u_1)$ decreasing in i. We know that $L(i)$ is nondecreasing in i from Theorem 1. Thus, from Equation (A1), we get that $D_{13}(i)$ is decreasing in i. By a similar argument, and because P is upper triangular and IFR matrix, we can show that $D_{23}(i)$ is decreasing in i.

As discussed earlier in §3, the initial cost of a major upgrade will be substantial. Hence $D_{13}(i)$ and $D_{23}(i)$ will be positive in the initial stages after the release of a new system. Because $D_{13}(i)$ and $D_{23}(i)$ are decreasing in i, we denote the first state where both $D_{13}(i)$ and $D_{23}(i)$ are below zero as the state i^*. That is, for any state higher than i^*, it will be optimal to do a major upgrade to the software system. Q.E.D.

COROLLARY 1. *The optimal major upgrade state of the system i^* cannot decrease (cannot increase) if the major upgrade cost function* $\text{RC}(i)$ *is replaced with another function* $\text{RC}^d(i)$ *such that* $\text{RC}^d(i)$ *is greater (less) than* $\text{RC}(i)$ *for all i, and satisfies all the assumptions on* $\text{RC}(i)$.

PROOF. Let $D^d_{13}(i)$ and $D^d_{23}(i)$ be the same functions as $D_{13}(i)$ and $D_{23}(i)$ defined in the proof of Theorem 2 using the major upgrade cost function $\text{RC}^d(i)$. Because $\text{RC}^d(i)$ satisfies all the assumptions on $\text{RC}(i)$, $D^d_{13}(i)$ and $D^d_{23}(i)$ are decreasing in i from Theorem 2. Note that if $\text{RC}^d(i)$ is less (greater) than $\text{RC}(i)$ for all i, by definition $D^d_{13}(i)$ and $D^d_{23}(i)$ are less (greater) than $D_{13}(i)$ and $D_{23}(i)$ respectively. The result follows by definition of i^*. Q.E.D.

PROPOSITION 1. *If the system is changed with a major upgrade using a superior technology such that the new technology deteriorates slower (i.e., $R = P^o - P^n$ is monotonically increasing in each row), and costs for major and minor upgrades, and operational decisions are lower for the superior technology, then* (1) *the optimal discounted cost for using superior technology starting from any state i cannot exceed that for the current technology; and* (2) *the optimum first major upgrade state i^* using the superior technology is lower than that for the current technology.*

PROOF.

(i) Let $L^n(i)$ be the optimal discounted cost for the superior technology starting at any state i, and let $L^o(i)$ be the same cost for the current technology. Similar to operator K in Equation (5) in Theorem 1, we define K_n and K_o as the

two operators in the new superior and current technology, respectively. Similar to the logic in the proof of Theorem 1, using Equation (7) and the condition that R is monotonically increasing in each row, it can be shown by mathematical induction that

$$K_n^q(i) < K_o^q(i) \text{ for all } i \text{ and } q \text{ and consequently}$$
$$L^o(i) \geq L^n(i) \text{ for all } i. \tag{A3}$$

(ii) Let $D_{13}^n(i)$ and $D_{23}^n(i)$ be the two difference functions as defined in Theorem 2, when the option of major upgrade with a superior technology is considered. Then we get that

$$D_{13}^n(i) = \mathrm{SC}^n(i, u_3) - \mathrm{SC}^o(i, u_1) + \delta L^n(0) - \delta L^o(i)$$

$$D_{23}^n(i) = \mathrm{SC}^n(i, u_3) - \mathrm{SC}^o(i, u_2) + \delta L^n(0) - \delta \sum_{j=0}^{Z} p_{ij} L^o(j).$$

In the above equations, it is reflected that the new superior technology is used from State 0 after the major upgrade. From the first part of this proposition, we know that $L^o(i) \geq L^n(i)$ for all i. Hence we get from Equation (A1) and (A2) that $D_{13}^o(i) \geq D_{13}^n(i)$, and $D_{23}^o(i) \geq D_{23}^n(i)$. Thus, from Theorem 2 the optimal state i^* when the superior technology is considered is lower. Q.E.D.

Corollary 3. *In the presence of a partial rewrite option for the dm, there exists a state g^* beyond which it is always optimal to do major upgrade to the system.*

Proof. Because we assume a large value Δ for $\mathrm{SC}(i, u_4)$ when $i < k$, the option u_4 will never be optimal for any state i, when $i < k$. Similar to difference functions $D_{13}(i)$ and $D_{23}(i)$ in Theorem 2, we define

$$D_{43}(i) = \mathrm{SC}(i, u_3) - \mathrm{SC}(i, u_4) + \delta L(0) - \delta L(k). \tag{A4}$$

From Assumption 5, we know that $D_{43}(i)$ is decreasing in i for $i > k$. Because $D_{13}(i)$ and $D_{23}(i)$ are decreasing in i, we denote the first state where $D_{13}(i)$, $D_{23}(i)$, and $D_{43}(i)$ are negative as the state g^*. Q.E.D.

References

Banker, R. D., S. Slaughter. 1997. A field study of scale economies in software maintenance. *Management Sci.* **43**(12) 1709–1725.

Banker, R. D., S. M. Datar, C. F. Kemerer. 1991. A model to evaluate variables impacting the productivity of software maintenance projects. *Management Sci.* **37**(1) 1–18.

Banker, R. D., S. M. Datar, C. F. Kemerer, D. Zweig. 1993. Software complexity and software maintenance costs. *Comm. ACM* **36**(11) 81–93.

Barua, A., T. Mukhopadhyay. 1989. A cost analysis of software dilemma: To maintain or to replace. *Proc. 22nd Hawaii Internat. Conf. Systems Sciences*, 34–43.

Bertsekas, D. 1987. *Dynamic Programming and Stochastic Control*. Academic Press, New York.

Booch, G. 1993. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, CA.

Chan, T., S. L. Chung, T. H. Ho. 1996. An economic model to estimate software rewriting and replacement times. *IEEE Trans. Software Engrg.* **22**(8) 580–598.

Chand, S., T. McClurg, J. Ward. 1993. A single machine replacement with learning. *Naval Res. Logist.* **40** 175–192.

Dean, D. L., R. E. Dvorack, E. Holen. 1994. Breaking through the barriers to new systems development. *McKinsey Quart.* **3** 3–13.

Derman, C. 1963. On optimal replacement rules when changes of state are Markovian. R. Bellman, ed. *Mathematical Optimization Techniques*. University of California Press, Berkeley, CA, 201–210.

Gelman, S. J., F. M. Lax, J. F. Maranzano. 1992. Competing in large-scale software development. *AT&T Tech. J.* (November/December) 2–10.

Gode, D. K., A. Barua, T. Mukhopadhyay. 1990. On the economics of the software replacement problem. *Proc. 11th Internat. Conf. Inform. Systems*, 159–170.

Hoffman, D. J., J. F. Rockart. 1994. Application templates: Faster, better and cheaper systems. *Sloan Management Rev.* (Fall) 49–60.

Jones, C. 2000. *Software Assessments, Benchmarks and Best Practices*. Addison-Wesley Information Technology Series, Boston, MA.

Keith, B. 1995. Legacy systems: Coping with success. *IEEE Software* **12**(1) 19–23.

Kemerer, C. F. 1987. Measurement of software development productivity. Unpublished doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA. University Microfilms International, Ann Arbor, Michigan, 89-05252.

Kung, D. C., J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen. 1994. Change impact identification in object oriented software maintenance. *Proc. Internat. Conf. Software Maintenance*, Victoria, B.C., Canada, 202–211.

Lehman, M. M., L. A. Belady. 1985. *Program Evolution: Process of Software Change*. Academic Press, San Diego, CA.

Linger, R. C. 1993. Cleanroom software engineering for zero defect software. *Proc. 15th Internat. Conf. Software Engineering*, Baltimore, MD.

Nurani, R. K. 1995. Role of in-line process monitoring and control in wafer fab management. Unpublished doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA.

Rosenfield, D. 1976. Markovian deterioration with uncertain information. *Oper. Res.* **24** 141–155.

Ross, S. M. 1971. Quality control under Markovian deterioration. *Management Sci.* **17** 587–596.

Rugaber, S., S. Doddapaneni. 1993. The transition of application programs from COBOL to a fourth generation language. *Proc. IEEE Internat. Conf. Software Maintenance*, 61–69.

Schneidewind, N. 1987. The state of software maintenance. *IEEE Trans. Software Engrg.* **13**(3) 303–310.

Sethi, S. P., S. Chand. 1979. Planning horizon procedures in machine replacement models. *Management Sci.* **25** 140–151.

Sneed, H. M. 1995. Planning the reengineering of legacy systems. *IEEE Software* **12**(1) 24–34.

Sneed, H. M., E. Nyary. 1994. Downsizing large application programs. *J. Software Maintenance* **6**(5) 235–247.

Swanson E. B. 1976. The dimensions of maintenance. *Proc. 2nd Internat. Conf. Software Engrg.*, San Francisco, CA, 492–497.

Swanson, E. B., C. M. Beath. 1989. *Maintaining Information Systems in Organizations*. John Wiley & Sons, New York.