SERC

Software Engineering
Research Center

A Formal Model of the Software Test Process

João W. Cangussu

Raymond A. DeCarlo

Aditya P. Mathur

Technical Report: SERC-TR-176-P

March, 2001

A Formal Model of the Software Test Process

João Cangussu [*]
Deptartment of Computer Science
Purdue University
West Lafayette - IN
47907-1398, USA
(765)494-7812
cangussu@cs.purdue.edu

Raymond A. DeCarlo
Deptartment of Electrical and Computer Engineering
Purdue University
West Lafayette, IN
47907-1285, USA
(765)494-3523
decarlo@ecn.purdue.edu

Aditya P. Mathur[†]
Deptartment of Computer Science
Purdue University
West Lafayette - IN
47907-1398, USA
(765)494-7823
apm@cs.purdue.edu

Abstract

A novel approach to model the system test phase of the software life cycle is presented. This approach is based on concepts and techniques from the theory of automatic control and is useful in computing the effort required to reduce the number of errors and the schedule slippage under a changing process environment. Results from these computations are used, and possibly revised, at specific checkpoints in a feedback-control structure to meet the schedule and quality objectives. Two case studies were conducted to study the behavior of the proposed model. One study uses data from the error log reported by Knuth while the other from a commercial project. The outcome from these two studies suggests that the proposed model might well be the first significant milestone along the road to a formal and practical theory of software process control.

Keywords: Feedback control, process control, Software test process, software testing, modeling, state variable.

1 Introduction

Research in software process modeling dates back to the early seventies. A detailed account of its evolution is given by Ghezzi [5]. In this account, the features of PROSYT, a second generation Process-centered Software Engineering Environment, are grouped by Ghezzi into three main areas: (i) process modeling, (ii) process enactment, and (iii) system architecture. Under process enactment, Ghezzi states: "To better control process execution, PROSYT allows process managers to specify a *deviation handling* and a *consistency checking* policy. Such policies state the level of

[*] Financial support by CAPES and UFMS
[†] Financial support in part by SERC and NSF

enforcement adopted ... and the actions that have to be performed when the invariants are violated as a result of deviation, respectively.” The problem of “managing unforeseen situations,” also referred to as “tolerating deviations,” is formulated and solutions proposed by Cugola [4]. Cugola has also proposed policies to handle various types of deviations. Though useful in practice, the policies do not assist in the computation of quantitative values of corrections that are often needed when deviations occur in process variables that can be, and often are, measured in numerical terms; project schedule and product quality are examples of such process variables.

Ghezzi et al. [11] explain the need for and the relationship between rigor and formality. They state “Also, various degrees of rigor can be achieved. The highest degree is what we call *formality.* Thus, formality is a stronger requirement than rigor; it requires the software process to be driven and evaluated by mathematical laws.” Our belief, supported by the vast amount of literature, is that research since the 70’s has focused more on obtaining added rigor than on bringing formality to the control of software processes. Contents of Ghezzi’s work [5], and those of most citations therein, also seem to support our belief.

We are aware of formal approaches to software process modeling, e.g. state models of software processes. Statistical process control is another formal approach to process control. However, in practice these and other less formal but rigorous approaches fall far short of the formalisms for process control that exist in other engineering disciplines. For example, the theory of automatic control is rich in formalisms that are practical and in regular use in chemical process control. Temperature control, aircraft wing control, continuous gravimetric control, are only a few examples of a myriad of control applications in the industry that rely on control algorithms firmly grounded in theory. One of the issues the theory deals with in a formal manner is the reduction of deviations from one or more set points characteristic of the process under control; e.g. temperature in a boiler control system.

Research reported herein is perhaps the first step towards a formal theory of software process control. The long term objective of this research is to borrow, adapt, and modify when needed, from the rich theory of automatic control; especially from the theory of state variables. For the short term, we decided to focus our efforts to one significant phase of the Software Development Process (SDP), namely the Software Test Process (STP). Though control of other phases of the SDP is often as important to an organization as the control of the test phase, the following two reasons motivated us to select the STP: (1) STP lends itself well to the characterization of input, output, and internal process variables and (2) there is a significant amount of data available from past and ongoing projects that is a key to the conduct of case studies to investigate the applicability of our model and approach.

Certainly, the problem of identifying and estimating the key parameters to be included in any model of the STP is difficult and discussed at length in this paper. Also, testing occurs at different points during the software life cycle. We focus on the system test phase which we assume occurs when a product is ready to be tested. There are variations on how the system test phase is conducted and its relationship to the other phases of the life cycle. Here we focus on one kind of system test phase where the test and debug cycles alternate *not* based on a predetermined schedule but as and when the need arises.

The remainder of this paper is organized as follows. In Section 2 we state a process control problem that arises within the context of the STP. This problem is by no means the only control problem that one might need to deal with. It is, however, the focus of our attention in the rest of this paper. In Section 3 we offer an introduction to feedback control from the point of view of a software engineer. The modeling approach and the underlying motivation appear in Section 4 which also describes a model based on the use of state variables. Estimation of various parameters of our model is discussed in Section 6. The behavior of the model under extreme conditions is analyzed in Section 5. This analyses is the first step towards assessing the accuracy of the model. Two case studies to analyze the behavior of the model are presented in Section 7. A description of other modeling approaches and a comparison of these with ours is presented in Section 8. Finally, in Section 9 we present our conclusions and outline directions for future work in the area of process modeling using the theory of feedback control.

2 The STP control problem and its context

2.1 The context

The key components of an SDP are exhibited in Figure 1. We focus on the system test phase. The unit and integration test phases are not accounted for in our model. We focus on the control of the time and effort required to reduce the errors by a desired fraction. Thus, upon the completion of coding and unit testing, any effort to test and debug is considered in our model. Such testing often occurs as the final phase before the application is delivered to the customer for beta testing or for actual use. Though various phases of the STP can occur concurrently, we assume that such concurrency affects only the effort applied during the execution of a phase. Also, though inspections can be, and often are, performed after each phase of the SDP, we do not consider inspections as part of the STP.

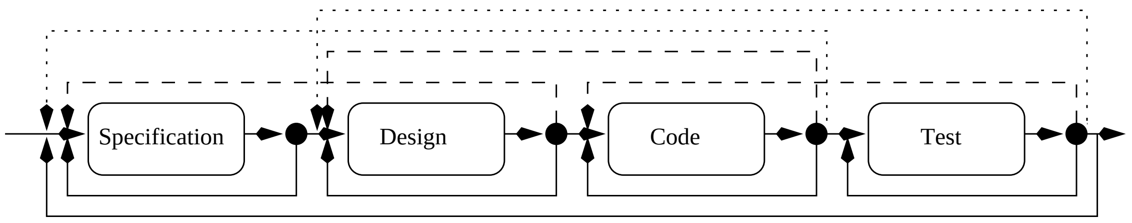

Figure 1: Different phases in a software development process.

2.2 The STP control problem

To explain the problem of control of an STP, consider an application $\mathcal{P}$ under test. We assume that the quality and schedule objectives are set at the start of the test process. The quality objective might be expressed in a variety of ways. For example, it might be expressed in terms of the reliability of $\mathcal{P}$ at the end of the test process. It might also be expressed in terms of the number of errors remaining at the end of the test phase. The schedule might be expressed in terms of a target date or, more elaborately, in terms of a sequence of checkpoints leading to a target date. Further, the quality objective could be more refined and specified for each checkpoint.

We assume that a test manager plans the execution of the test phase to meet the quality and schedule objectives. Such a plan involves several activities including the constitution of the test team, selection of the test tools to use, identification of training needs and scheduling of training sessions. Each of these activities involves estimation. For example, constituting the test team requires a determination of how many testers to use. The experience of each tester is another important factor to consider. It is the test team that carries out the testing activity and hence spends the effort that will hopefully help in meeting the objectives. The ever limited budget is usually a constraint to contend with. During the initial planning phase, a test manager needs to answer the question, "How much effort is needed to meet the schedule and quality objectives?" Experience of the test manager does help in answering this question. However, we approach this problem from a mathematical standpoint.

The question stated above is relevant at each checkpoint. We assume that the test manager has planned to conduct reviews at intermediate points between the start and the target date of the STP. The question might arise at various other points also, for example when there is attrition in the test team. An accurate answer to the above question is important not only for continuous planning but also for process control. A question relevant for control is: "How much additional test effort is required at a given checkpoint if a schedule slippage of, say, 20% can be tolerated?" This question could be reformulated in many ways in terms of various process parameters. A few other related questions of interest to a test manager are enumerated below.

1. Can testing be completed by the deadline and the quality objective realized?

2. How long will it take to correct the deviations in the test process that might arise due to an unexpectedly large number of reported errors, turnover of test engineers, change in code, etc.?

3. By how much should the size of the test team be increased if the deadline is to be advanced without any change in the error reduction objectives?

To answer the above questions, we propose a model based on the application of the theory of feedback control using a state variable representation. Our model allows comparison of the output variables of the software test process with one or more "setpoint(s)." Such a comparison leads to the determination of how the process inputs and internal parameters ought to be regulated to achieve the desired objectives of the STP.

2.3 The test-debug cycle

There are a variety of ways to organize the system test phase. The action taken by a test team upon the failure of $\mathcal{P}$ on one or more inputs could result in variations in the sequencing of activities within system test phase. For example, a test team might decide to send $\mathcal{P}$ back to the development team when one or more failures of a critical nature are observed. This might lead to a temporary suspension of the system test phase. In contrast, a test team might decide to complete the system test by running all tests as planned, and then send $\mathcal{P}$ back to the development team if a sufficiently large number of failures were detected. In yet another scenario, the test and the development team could work concurrently.

While modeling the STP, we decided to assume that the test-debug cycle occurs as shown in Figure 2. According to this figure, a $\mathcal{P}$ could be in one of three states: *Test*, *debug*, and *End*. When in *Test*, $\mathcal{P}$ is executed until a failure is detected. At this point if a debug-mode condition is true then $\mathcal{P}$ enters the *debug* state, otherwise it remains in *Test*. The debug-mode condition could be, for example, "The number of failures detected exceeds a threshold." As another example, it could be "A failure that will prevent succeeding tests to run as planned." $\mathcal{P}$ remains in the *Debug* state until errors are found and fixed when it returns to the *Test* state. However, while $\mathcal{P}$ is in *Debug*, another avatar of it could remain in the *Test* state. This is when we say that testing and debugging are taking place simultaneously.

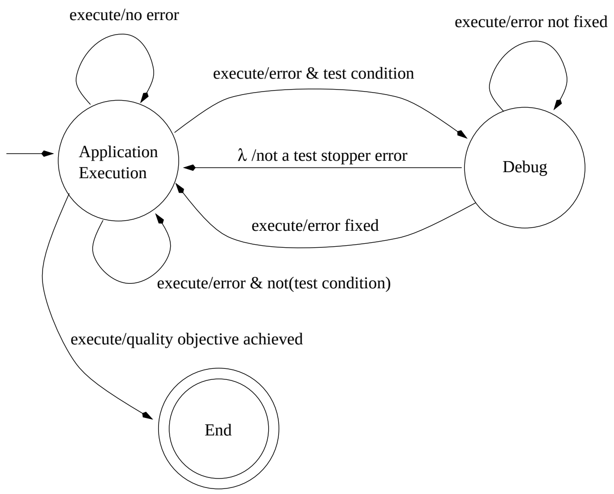
execute/no error
execute/error not fixed
execute/error & test condition
Application Execution
λ /not a test stopper error
Debug
execute/error fixed
execute/error & not(test condition)
execute/quality objective achieved
End

Figure 2: A test-debug cycle.

Though the test-debug cycle described above is handled effectively by our model, other possibilities could also be handled with some care in collecting the data needed for using the model. A few such possibilities are discussed later in Section 9.

3 Feedback control in the context of the Software Test Process

When applied to the STP, the objective of feedback control is to assist a test manager in making decisions regarding the expansion of or reduction in workforce and the change of the quality of the test process. The control process is applied throughout the STP. Though it does not guarantee that the STP will be able to meet its schedule and quality objectives, it does provide information that helps the test manager determine whether or not the objectives will be met and, if not, what actions to take.

We assume that the schedule objective is specified as: *Complete the test process by a specified time t_f.* The quality objective is specified as: *Ensure that at most r_f errors remain in $\mathcal{P}$ at time t_f.* The quality objective could be stated in several other ways. In our work we assume that any reduction in software errors that remain in a product improves the quality of that product. In the work that follows do not distinguish among the various types of errors such as specification errors, critical and non-critical errors, etc.

Hence we specify the quality objective in terms of the number of remaining errors. The difficulties in estimating the number of remaining errors are overcome using techniques described in Section 6. We are also aware that an STP might be driven by objectives other than, or in addition to, the two specified here. However, for the purpose of the control of STP, our current focus is on schedule and quality.

We also assume that prior to the start of the STP, the project manager sets up a monitoring schedule that consists of a sequence of $k, k > 0$ checkpoints over time. The i^{th} checkpoint, denoted by cp_i, is specified as time t_i when monitoring is to take place and r_i, the number of errors expected to remain in $\mathcal{P}$ at time t_i. The first checkpoint occurs some time after the start of the STP, i.e. $t_1 > 0$, and the last checkpoint coincides with the deadline. Thus, for the k^{th} checkpoint, $t_k = t_f$ and $r_k = r_f$. The economics of a software project will most likely constrain its budget. The budget is not included explicitly in our model. However, the proposed feedback control mechanism assists a project manager in tracking possible budget overruns. Also, a project manager need not explicitly specify r_i. Instead, the specification could be in terms of a fraction by which the number of errors is expected to be reduced. Thus, for example, this fraction could be $0 < f_i < 1$ at the i^{th} checkpoint. This would imply that the number of errors expected to remain in $\mathcal{P}$ at checkpoint cp_i is $f_i \times r_{i-1}$.

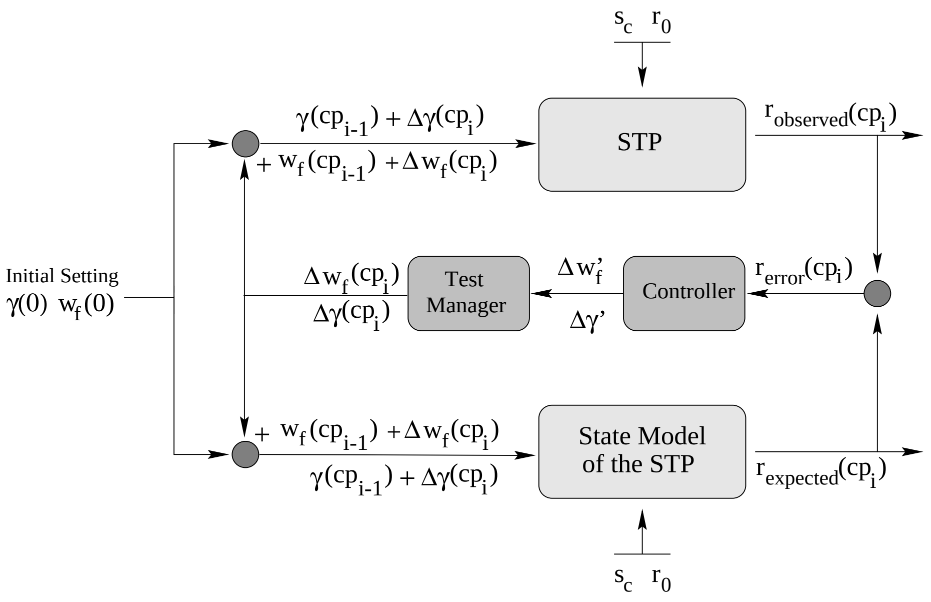

Figure 3: Feedback control of the Software Test Process.

The use of feedback control can be understood from Figures 3 and 4. The continuous line in Figure 4 shows the expected variation in $r(t)$ over the course of the STP; $r(t)$ is computed using the state model described in Section 4.4. As shown by the dashed lines, the test manager uses this prediction to generate a schedule in terms of the checkpoints. The checkpoints are determined by the test manager and the expected values of the number of remaining errors at each checkpoint, denoted by $r_{expected}(\mathrm{cp}_i)$ at checkpoint cp_i, can be read off the $r(t)$ function. The test process is started at time $t = t_0$ at which point $\mathcal{P}$ contains $r = r_0$ errors.

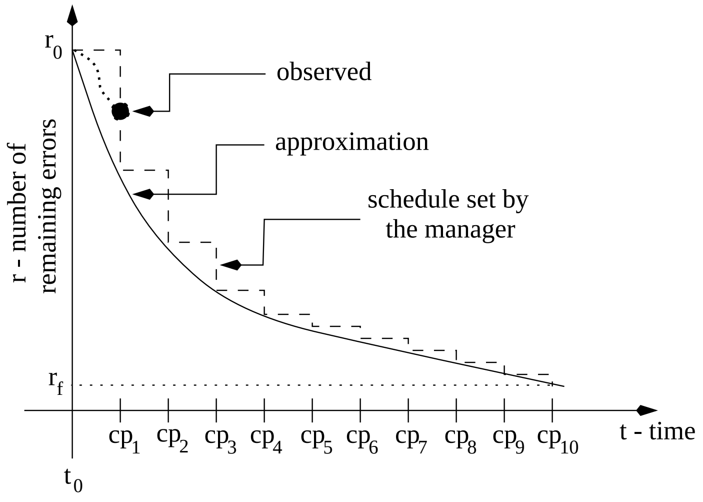
r0
observed
approximation
schedule set by the manager
r - number of remaining errors
rf
cp1 cp2 cp3 cp4 cp5 cp6 cp7 cp8 cp9 cp10
t - time
t0

Figure 4: Checkpoints in a Software Test Process. cp_i denotes the i^{th} checkpoint.

At checkpoint cp_1 the observed value of the number of remaining errors, denoted by $r_{observed}$, is compared with $r_{expected}$ to generate the input $r_{error}(\mathrm{cp}_i)$ to the controller. The controller computes the changes needed to the workforce, $\Delta w'_f$, and the quality of the test process, $\Delta\gamma'$ for the objectives to be met. The test manager uses the changes computed by the controller to decide whether or not any change is needed in the test process. Thus, for example, the test manager might decide to ignore the controller output. In this case the STP continues until the next checkpoint without any changes in w_f and γ. However, the manager might decide to make use of the output from the controller and change only the workforce to $w_f + \Delta w_f$ and keep γ at its current value. Obviously, several possibilities exist. In any case, the STP continues with the workforce and process quality set to, respectively, $w_f + \Delta w_f$ and $\gamma + \Delta\gamma$. Note that these updated values of the workforce and the quality of the test process are also input to the model which in turn re-computes $r(t)$.

The process described above continues until the STP is completed. During this process, the manager might decide to change the checkpoints and even the objectives of the STP. In any case, the STP model and the controller are provided with the updated values and generate useful data throughout the STP. Of course, there is no guarantee that there exists a feasible solution to the problem of completing the STP with the desired objective being met. However, if a feasible solution exists, then the model finds it and assists the test manager in steering the STP.

4 Modeling the Software Test Process

4.1 Variables and parameters

A number of variables and parameters are specific to the system test phase. We consider the following:

1. (r) - the number of remaining errors.
2. (w_f) - size of the test team.
3. (s_c) - program complexity.
4. (t) - time measured in appropriate units.
5. (γ) - a constant characterizing the overall quality of the test process.
6. (e_f) - effective test effort.
7. (e_r) - error reduction resistance.

One could use any one of the existing complexity measures to obtain a value of s_c. Our model does not prescribe any specific complexity measure. One could use, for example, program size, cyclomatic complexity [27] or a combination of both to compute s_c.

The coefficient γ characterizes the overall quality of the test process and represents environmental factors such as pressure due to deadline, test methodology used, structure of the organization within which testing is carried out, experience and expertise of members of the test team, and possibly other factors. Although a single coefficient is unable to fully represent the quality of the testing phase, when appropriately chosen it appears to be adequate.

The effective test effort (e_f) is the actual effort expended by the test team to reduce the number of errors. The test team is often more successful at the beginning of the test phase in finding and removing errors than towards the later part. The team members also spend time on ancillary tasks to fulfill their administrative and reporting responsibilities and to learn the use of new software tools. Such tasks tend to decrease the effective test effort.

The process of error removal might cause the introduction of new errors in the application under test. This results in wasteful effort by the test team. We use the error resistance, e_r, to model this wasted effort. e_r is considered as effort that tends to oppose the effect of the effort applied to remove the errors.

4.2 Key assumptions

Next we present three key assumptions about the system test phase. These assumptions lead to the fundamental laws that we believe, and justify, govern the STP. The fundamental laws, and the resulting equations, lead to a state model of the STP. It is the solution of this state model that allows a test manager to answer schedule related questions mentioned earlier in Section 1.

Assumption 1

The rate at which the speed of decrease of the remaining errors changes is directly proportional to the net applied effort and inversely proportional to the complexity of the program under test.

Formally, this assumption is restated in Eqn. 1.

$$\ddot{r} = \frac{e_n}{s_c} \quad \Rightarrow \quad e_n = \ddot{r}\ s_c \tag{1}$$

where $\ddot{r}$ denotes the second derivative of r and e_n the net applied effort.

The first assumption is justified as follows. When the same metric or combination of metrics is used to compute software complexity for two different programs under test, it is reasonable to expect that more effort will be necessary to test the more complex program. If, for example, Cyclomatic Complexity [23] and LOC are used to determine s_c, a larger program with more regions will likely require more test effort than a smaller program with a small number of regions.

The net applied effort (e_n) is the balance of all the effort applied during the test phase. This results from the difference of the effective effort applied by the test team minus any "frictional" forces that decrease the applied effort. Since r represents the number of remaining errors, its first derivative $\dot{r}$ is the error reduction velocity (v_e). Consequently, $\ddot{r}$, which denotes the rate of change of $\dot{r}$, is an acceleration. Thus, the concepts of velocity and acceleration have counterparts in the test phase.

In the world of software, Eqn. 1 is analogous to Newton's second law of motion for physical systems where e_n is analogous to physical force, $\ddot{r}$ to acceleration, and s_c to mass. In the physical world a larger mass requires a greater force to move it a given distance at a desired velocity. In the world of software, higher program complexity requires larger effort to reduce errors by a given fraction at a desired error reduction velocity ($\dot{r}$).

It is widely believed that the difficulty of finding program errors increases as the test phase progresses. Assuming a fixed team size, this implies that the effective test effort is directly proportional to r. This observation suggests Assumption 2.

Assumption 2

The effective test effort is proportional to the product of the applied work force and the number of remaining errors.

This assumption is represented formally by the following equation

$$e_f = \zeta\ w_f\ r \tag{2}$$

for an appropriate ζ.

Justification of this assumption follows by analogy with the predator-prey system described by Volterra [22]. Here, the decline in the prey population is proportional to the number of possible encounters between the predators and the prey, i.e. the product of the populations of predators and prey. Assumption 2 above presents similar characteristics to this widely accepted model. The probability of finding an error is equivalent to an encounter between a tester and an error. The tester plays the predator role and errors are the prey. There are $w_f\ r$ possible encounters. The parameter ζ defines the decline rate and it may decrease as r gets smaller ($\zeta = \zeta(r)$). That is, as the test process continues, the errors become more difficult to find, not only because there are less of them but also because some errors require a combination of events to be triggered and it is most likely this combination will be discovered by the testers only in the final phases of testing, if it is discovered at all. This behavior is captured by changes in ζ over different periods of the STP.

Assumption 2 can be understood with another analogy. In a spring the restoring force is determined by the spring stiffness and by how much the spring is extended beyond its natural length. Increasing the spring stiffness or the extension increases the restoring force. The effective test effort can be interpreted in an analogous way. The number of remaining errors is analogous to the spring length. At the beginning of the test phase r is larger than it is towards the end. Hence, the effective effort decreases as r decreases. The work force can be related to the spring stiffness. The larger the work force, the greater the restoring force, i.e., the effective effort. Thus spring stiffness is analogous to w_f and spring extension to the number of errors (r) already found in the application. In Eqn. 2, ζ remains constant over a period and must be calibrated for the project under analysis. The behavior of Assumption 2 is similar to the rate of decrease of errors [16, 29, 31] when software reliability models are applied to the STP [7].

The effective test effort is opposed by a force intrinsic to the test phase process. This force is the error reduction resistance, e_r. This observation leads to the last of the three assumptions.

Assumption 3

The error reduction resistance opposes and is proportional to the error reduction velocity and inversely proportional to the overall quality of the test phase.

Assumption 3 is represented formally in Eqn. 3.

$$e_r = -\xi\ \frac{1}{\gamma}\ \dot{r} \tag{3}$$

for an appropriate constant ξ. The negative sign indicates that the error reduction always opposes $\dot{r}$.

The assumption above can be justified by analyzing its behavior under extremal conditions. For example, a very low quality will induce a large resistance: $\gamma\ \rightarrow\ 0\ \ \Rightarrow\ \ e_r\ \rightarrow\ \infty$. The same is true for values of $\dot{r}$: the larger is $\dot{r}$, the larger is the error resistance e_r.

This assumption implies that the faster one tries to reduce the remaining errors the more likely one is to make mistakes which slows the entire process. A physical dashpot can be used to explain this behavior. The coefficient of viscosity of the liquid inside the dashpot is $\frac{1}{\gamma}$. Therefore, a small coefficient of viscosity is analogous to the test phase being conducted in a smooth and careful way and, thus, the number of new errors inserted is small. Larger coefficient of viscosity is analogous to the test phase in which more errors are introduced than would be introduced under normal circumstances. The velocity component in the dashpot is analogous to the error reduction velocity ($\dot{r}$). Thus, the overall quality of the test phase, denoted by (γ), and the rate at which errors are found, determines the error reduction resistance effort which is analogous to the damping force generated by the dashpot. In Eqn. 3, ξ is merely a constant of proportionality.

4.3 A differential equation model of the STP

Assumptions 2 and 3 relate to the test effort in our model while Assumption 1 to the resultant force balance. Combining the corresponding equations leads to the following force balance equation for the net effort:

$$-e_f \ + e_r \ = \ e_n \tag{4}$$

where e_f has a negative sign because it opposes the increase in errors. Replacing e_f, e_r and e_n by their values from Eqns. 1, 2 and 3 results in the following second-order differential equation:

$$-\zeta \ w_f \ r \quad -\xi \ \frac{1}{\gamma} \ \dot{r} \ = \ s_c \ \ddot{r} \tag{5}$$

As in our coordinate system we are applying a restoring force creating a velocity with a negative direction. Thus we thus find on the average that $\dot{r} < 0$. Hence, $|e_n| \ < \ |e_f| \ for \ \dot{r} < 0$. This leads to the following equation:

$$|-\zeta \ w_f \ r \quad -\xi \ \frac{1}{\gamma} \ \dot{r}| \ < \ |-\zeta \ w_f \ r| \tag{6}$$

An analogy to a dashpot system will clarify the behavior of Eqn. 5. Consider a physical system where a mass is attached to an extended spring and a dashpot. The spring is extended by 100 units and is attached to a wall at the other extreme as it is the dashpot. The spring restoring force will move the block from the initial position (100 units) to a position as close to zero as possible and the dashpot will retard this movement. Here we assume an overdamped system and hence the block will never reach a negative position. This behavior is analogous to what happens in the system test phase where the dashphot is related to e_r and the restoring force to e_f. By assumption, the system test phase starts with a program of complexity s_c and with 100% of remaining errors. The ideal goal is to remove errors until r approaches zero. The effective effort due to w_f and r, and the error reduction resistance due to γ and $\dot{r}$, will determine the rate of decrease of r. Table 1 summarizes our analogy between various elements of a physical system with those of the STP.

The analogy presented in this section is to clarify the behavior of our model. However, it is clear that the system test phase does not have exactly the same behavior as an arbitrary physical system. For example, if a spring is removed in a physical system, the inertia keeps the block moving. An analogous behavior during STP would imply that errors continue being removed from a program even if the workforce were reduced to zero. Certainly, this is not true at least in the current system test environments.

Table 1: Analogy between a physical and a software system.

Physical	Logical
Restoring force in a spring	Effective effort to find errors in a software
Spring stiffness	Work force applied to test phase and software complexity
Coefficient of viscosity	Overall quality of test phase
Block's mass	Software Complexity
Spring length	Remaining errors during test phase
Velocity	Error reduction velocity
Acceleration	Rate of change of error reduction velocity

Let r_0 denote the number of error remaining in the product at time $t = t_0$, i.e. at the beginning of the system test phase. The initial error reduction velocity is $v_e = 0$. Under this condition the solution to Eqn. 5 is:

$$r(t) = \frac{r_0 \lambda_2}{\lambda_2 - \lambda_1} e^{-\lambda_1 t} + \frac{r_0 \lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} \tag{7}$$

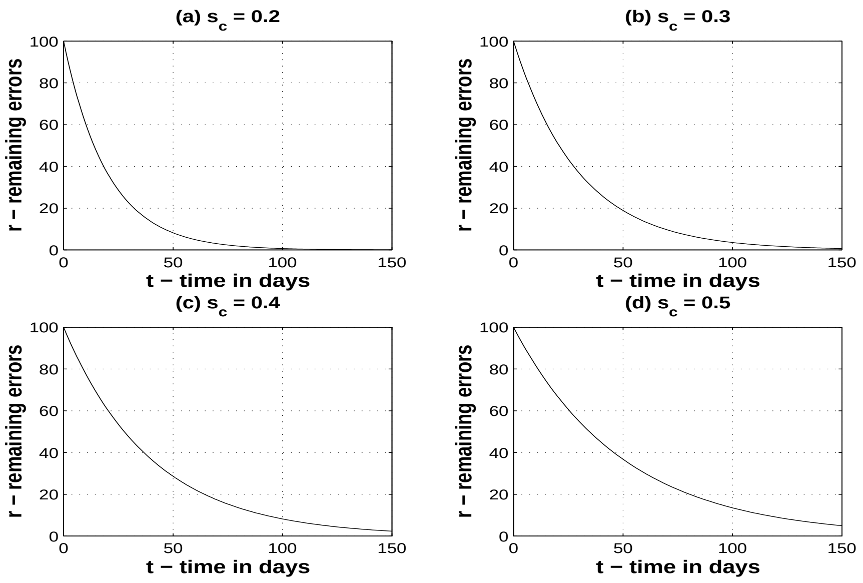

Figure 5: Effect of software complexity on the number of remaining errors as a function of time.

where λ_1 and λ_2 denote the distinct roots of the characteristic equation of Eqn. 5 [6, 22]. The two distinct negative roots are due to the assumption of a stable overdamped process. If underdamping were allowed, r would reach a negative value. This does not make sense in the world of software. The overdamping requirement restricts the values of γ to less than $\frac{\xi}{2(s_c\ w_f\ \zeta)^{\frac{1}{2}}}$ [10]. This is calculated by forming the characteristic equation and requiring the discriminant of the associated quadratic formula to be non-negative.

The behavior of Eqn. (5) is depicted in Figure 5 where one observes the results for $s_c = 0.2$, 0.3, 0.4 and 0.5; initial remaining errors of 100% ($r_0 = 100\%$); and initial error reduction velocity of 0 ($v_e = 0$). As expected, the number of remaining errors decreases slowly when system complexity is high, assuming that the overall quality (γ) and the work force (w_f) are held constant.

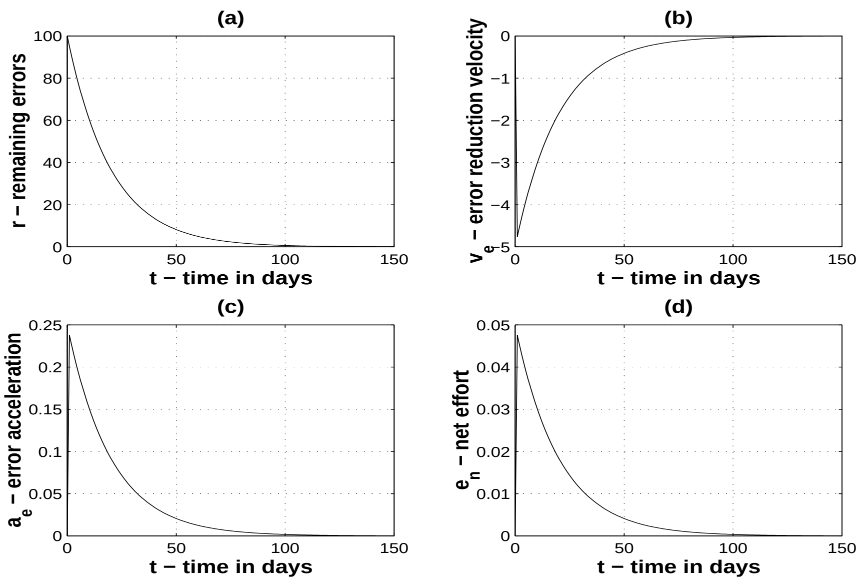

Figure 6: Number of remaining errors, reduction in the number of remaining errors, rate of reduction in the number of remaining errors for $s_c = 0.2$, $\gamma = 0.005$ and $w_f = 2$.

¿From Figure 6 we note how the error reduction velocity and the error acceleration change with a decrease in the number of remaining errors. Errors are easy to find at the beginning of an STP and therefore the velocity is relatively high. As the STP progresses, error detection becomes increasingly difficult and hence the velocity decreases to zero as observed in Figure 6b. Therefore, we have a deceleration until the chance of finding a new error becomes almost zero as in Figure 6(c). As expected according to Assumption 1, the net applied effort approaches zero as $t \to \infty$. This is depicted in Figure 6(d).

In Figure 6 we observe the results from a determined test team ($w_f = 2$) and in Figure 7 we observe the results of increasing the test team size by 1 while maintaining the same conditions. That is, in each of these cases we keep s_c and γ constant and vary only the team size.

In our model we consider only two forces acting on the program under test: the effective test effort (e_f) and the error resistance (e_r). However, there are other forces that affect a program during the STP. Hence, it is wise to include in the model forces represented by the auxiliary effort when a test tool is being used and also an opposite force when the test team spends time to learn the use of a new test tool during the STP. Other forces, not considered in this paper, include the communication effort and an adaptation effort.

Because our model captures the dominant dynamics of the STP, leaving certain forces as de-

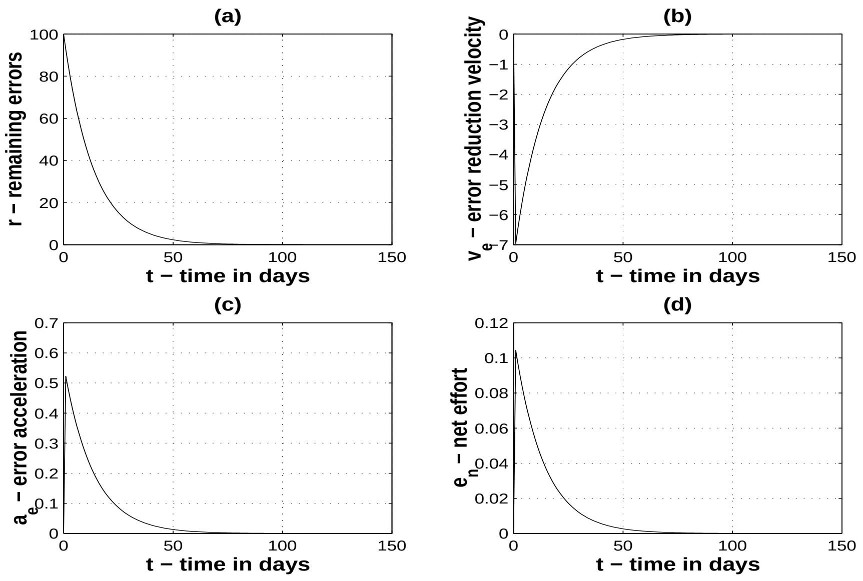

Figure 7: Number of remaining errors, reduction in the number of remaining errors, rate of reduction in the number of remaining errors, and the net applied effort for $s_c = 0.2$, $\gamma = 0.005$ and $w_f = 3$.

scribed above to be represented by a more complex future model, some error between the model predicted behavior and the observed behavior may exist. In addition, the STP is often beset by disturbances that may cause a delay in the process. For example, suppose a test team is using a populated database to test the product and for some reason the database becomes unavailable for 1 working day, making the team unable to test the program for the entire day. This would constitute a 100% disturbance if the time unit were days and a 20% disturbance if the time unit were weeks. In all cases, the disturbance represents a force, say F_d, opposing the effective test effort, e_f.[1] Thus it can be seen as a possibly event dependent percentage of the effective test effort. Of course, as more elements of the STP are accounted for in the model, then the contribution of say learning and communication to F_d will diminish to zero. Nevertheless, incorporation of F_d into Eqn. 5 results in Eqn. 8 below:

$$\ddot{r} \quad = \quad -\frac{\zeta \ w_f}{s_c} r \quad -\frac{\xi}{\gamma \ s_c}\dot{r} \quad +\frac{1}{s_c}F_d \tag{8}$$

Eqn. 8 will be the differential equation on which the state model is based.

4.4 A State Model for the STP

The state model developed here assists managers in making decisions about the STP as shown in Section 7. The state model is a matrix differential equation in a vector of state variables which in our case are the remaining errors, r, and the error reduction velocity, $\dot{r}$. These state variables are sufficient to model the dominant dynamics of the STP and also serve as the output variables of interest. Hence, with r and $v_e = \dot{r}$ as state variables, the following controllable canonical state model [6] results from Eqn. 8.

$$\begin{bmatrix} \dot{r} \\ \ddot{r} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{\zeta\ w_f}{s_c} & -\frac{\xi}{\gamma\ s_c} \end{bmatrix} \begin{bmatrix} r \\ \dot{r} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{s_c} \end{bmatrix} F_d \tag{9}$$

$$\begin{bmatrix} r \\ \dot{r} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r \\ \dot{r} \end{bmatrix} \tag{10}$$

where the matrix multiplying the vector of state variables is called the A-matrix. The model must be initialized at $t = 0$ which requires at least an estimate of $r(0) = r_0$ and the observation that the error reduction velocity $v_e(0) = \dot{r}(0) = 0$. The value of r_0 can always be updated as observed data becomes available over the course of the STP. Indeed data is needed to determine initial estimates for the proportionality constants ξ and ζ. Also, in this formulation, the workforce variable, w_f, is taken as a parameter in the A-matrix rather than an external input, such as F_d, because in the STP it is typically constant for a period or several periods of time before a manager may change it to a different level. For example, if $w_f = 5$ from time t_i to t_j, $i < j$, and changes to 6 from t_{j+1} to t_k, $(j + 1) < k$, we use $w_f = 5$ to observe the behavior for the first period and then switch to a system with $w_f = 6$ for the second period. To extract w_f from the A-matrix and make it an external input would move the model into the nonlinear category. By viewing w_f as a parameter, then the model remains in the piecewise-constant linear category and is amenable to well known solution techniques with the use of feedback as a parametric control to achieve management schedule objectives and reduce the impact of F_d. In addition, by having a piecewise-constant A-matrix, we may also update estimates for ξ and ζ.

4.5 Use of feedback control

In this section we show how feedback control can be applied to adjust STP parameters to meet the desired objective. The objective of an STP is restated below after combining the constraints on time to completion and the number of remaining errors.

Given that the program under test contains $r = r_0$ errors at the start of STP, it is desired to complete the STP in t_α weeks, such that the number of remaining errors r is reduced to $\alpha \times r_0$.

Once the STP objective has been set up, the project manager has two options. Option 1 is to organize a team of testers and start the STP. Under this option the manager does not estimate any of the model parameters and hence does not apply the model to check if the desired objective is indeed feasible.

Option 2 is to estimate the model parameters required to meet the objective and use the model to test if indeed the objective can be met. If the model indicates that the objective cannot be met with the estimated set of parameters, then another set is tried. This process continues until a reasonable set of parameter estimates is found at which point the STP is started. Note that only w_f and γ are under the control of the manager. Also, budgetary restrictions might impose additional constraints on these parameters. The choice of parameters is discussed in Section 6.

Prior to the start of the STP, a sequence of project review dates, also referred to as checkpoints, is decided upon. It is on these dates that the progress of the STP is reviewed and the model used to determine whether or not the originally stated deadline can be met. If not, then the model is used to determine the changes needed in the STP parameters in order to meet the deadline.

5 Extreme case analysis

We now subject the model in Eqs. 9 and 10 to an extreme case analysis with $F_d = 0$. The purpose is to understand how the model behaves under extreme conditions and whether or not this behavior is consistent with what one would expect of an STP under such conditions. We consider extreme conditions at the intersections of low and high values of software complexity and quality of the test phase. Note that it is inappropriate to analyze our model for the effects of extreme values of the work force because communication amongst testers is not included. Hence, for the purpose of this analysis we arbitrarily set $w_f = 5$. Table 2 shows the summary results of four extreme cases considered in this section.

The extreme case analysis proceeds as follows. For each of the four extreme cases identified in Table 2 we first compute s_c and set γ. These values, and that of w_f, are plugged into Eqn. 9 which is solved. The solution is depicted by a $r - t$ plot. From the plot we read-off $t_{0.05}$ which denotes the time needed in days to reduce the number of remaining errors to 5% of its initial value. Our assumption is that s_c and γ affect $t_{.05}$. We then compare the values of $t_{0.05}$ to determine the nature of this effect and compare it with what a software tester would expect intuitively.

Table 2: Summary of model behavior under extreme conditions.

Case	**Software Complexity** (s_c)	**Quality of the test phase**(γ)	**Time in days to reduce the number of errors to 5% of their initial value** ($t_{0.05}$).	**Figure**
1	0.875 (low)	0.05 (low)	52	8(a)
2	0.875 (low)	0.95 (high)	3	8(b)
3	30 (high)	0.05 (low)	1800	8(c)
4	30 (high)	0.95 (high)	95	8(d)

For all four extremal cases the parameter ξ was set to 100 as a factor of normalization. The parameter ζ cannot be estimated because the expected deadline is not available. Thus we set ζ to $20/s_c$ to represent the declining rate of ζ which is affected by software complexity.

For the purpose of our analysis, s_c is considered to be a convex combination of M_1, the lines of code measured in 10 KLOCs, and M_2 the average of Cyclomatic Complexity per function. The weights for M_1 and M_2 are set to, respectively, $\alpha_1 = 0.75$ and $\alpha_2 = 0.25$. Thus $s_c = \sum_{i=1}^{2} \alpha_i M_i$.

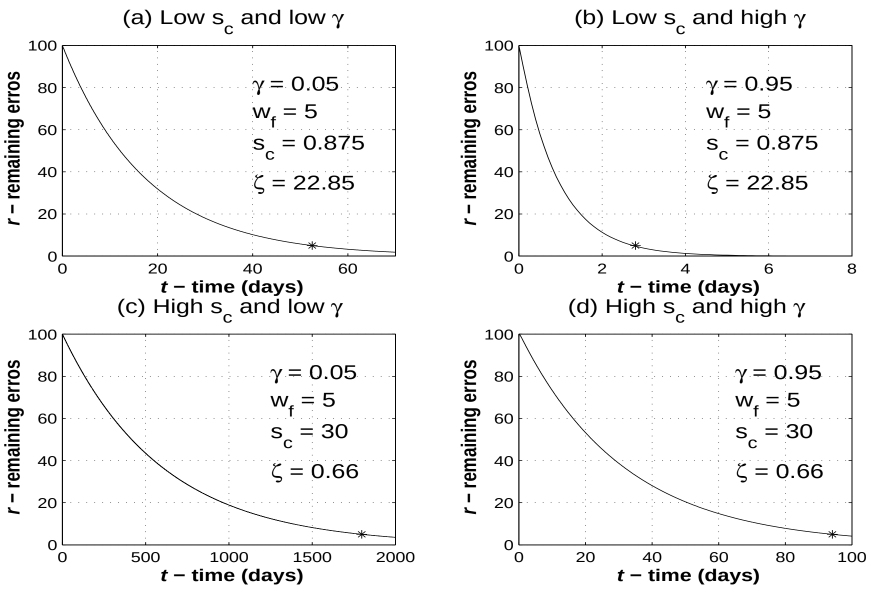

Figure 8: Variation in the number of remaining errors for four extreme cases based on s_c and γ.

Case 1: Low s_c and low γ

Consider a program with five thousand lines of code and having an average Cyclomatic Complexity of 2. Thus we have $M_1 = 0.75$, $M_2 = 2$ and hence $s_c = 0.5 * 0.75 + 0.25 * 2 = 0.875$. We quantify a low quality test phase by setting $\gamma = 0.05$. Substituting for parameters in Eqn. 9 and solving for r, we observe the behavior exhibited in Figure 8(a). As is evident from Figure 8(a), it requires 52 days to reduce the number of remaining errors in this product to less than 5%.

Case 2: Low s_c and high γ

Figure 8(b) represents the behavior predicted by our model for an almost perfect, though unrealistic, test phase. Some characteristics of such a test phase are: each member of the test team knows exactly what each part of the product does, can apply 100% of available time to the test effort, does not communicate with other members of the test team, and requires no learning once testing has begun. For this test phase we set $\gamma = 0.95$. We also consider a program containing fifty thousand lines of code with an average Cyclomatic Complexity of 2. This yields $s_c = 0.875$. These parameter values lead to a solution depicted in Figure 8(b). From this figure we observe that it will take about 3 days to reach the desired level of error reduction. As one might expect, as $s_c \rightarrow 0$ and $\gamma \rightarrow 1$ the time required to reduce the errors also diminishes and becomes proportional to the net applied effort.

Case 3: High s_c and low γ

Here we assume that the program under test contains three hundred thousand lines of code and has an average cyclomatic complexity of 30. This leads to $s_c = 30 * 0.75 + 0.25 * 30 = 30$.

We set $\gamma = 0.05$ to represent low quality. The solution to Eq 7 is plotted in Figure 8(c). The plot reveals that it will take about 1800 days for five testers to reduce the number of remaining errors to less than 5% of the initial count. As expected, this is considerably longer than in case 2. In general we can state that as $s_c \rightarrow \infty$ and $\gamma \rightarrow 0$ then $T \rightarrow \infty$, where T is the time required to reduce the errors to less than 5%.

Case 4: High s_c and high γ

In this case we assume a high quality test phase, as described in Case 2 and a high level of complexity as described in Case 3. The other parameters are set as before. Figure 8(d) reveals that the high quality of the test phase has a significant impact on the time required to reduce errors. In this case it will take about 94 days to reach the desired error reduction.

Summary of extreme case nalysis

Table 2 summarizes the behavior of r predicted by our model. This behavior is close to what a test engineer might expect under extreme conditions. The following common sense expectation of a test engineer is mimiced well by our model:

> *Complexity of the application under test, quality of the test team, and the appropriateness of the techniques used have a significant effect on the time to test.*

6 Estimating model parameters

The set of parameters listed in Section 4 is representative of the most significant aspects of the STP. Estimation of these parameter values is essential to a successful application of the state model. Some parameters are relatively easy to quantify while others are more subjective. Also, different organizations use different metrics and methodologies in the STP. There are no globally accepted metrics for the parameters and variables involved in the STP. Most models for the SDP rely on intuitively and/or empirically derived values for the parameters. This situation suggests a need for a methodology to help guide the estimation process. In the remainder of this section we discuss how one could estimate each of the several parameters needed to apply our model.

6.1 Size of the work force

The work force, w_f, is defined as the number of testers per unit time. As testers might be added or taken away as the STP progresses, any variation in w_f is accounted for by computing the state variables over successive periods.

6.2 Estimation of ξ, ζ, and r_0

The algorithm, described later in this section, is used for computing ξ, ζ, and r_0 from data from the current project. Obviously, this data is not available initially. However, a manager may have data from past similar projects that can be used to obtain initial estimates until data from the current project becomes available when the estimates can be improved.

The state model of Eqn. 9 has the general form of

$$\dot{x}(t) \quad = \quad Ax(t) \tag{11}$$

where $x(t) = [r(t)\ \dot{r}(t)]^T$ and A is the proper 2×2 matrix. It is well known that the solution of Eqn. 11 is given by $x(t) = e^{At}x(0)$ for all $t \geq 0$. To compute ξ and ζ we need to compute

$$A \quad = \quad \begin{bmatrix} 0 & 1 \\ -\frac{\zeta\ w_f}{s_c} & -\frac{\xi}{\gamma\ s_c} \end{bmatrix} \tag{12}$$

from which we can obtain ξ and ζ as all the other parameters are know at this time.

Initially $\dot{r}(0) = 0$ but $r(0)$ is not known. Further, project data ordinarily consists of the number of errors found and fixed in a time period of length, say T_1, i.e., project data consists of $d^{(k)} \equiv r(kT_1) - r((k-1)T_1)$. Although $r(t)$ has the general form specified in Eqn. 7, we use a single exponential approach to obtain a local approximation for $\dot{r}(t)$, i.e., we locally approximate $r(t)$ as

$$r(t) = \alpha e^{-\lambda t} \tag{13}$$

Hence for fixed T_1, let $m = \alpha e^{-\lambda T_1}$, then

$$d^{(k)} \quad \equiv \quad r(kT_1) - r((k-1)T_1) = mr((k-1)T_1) - mr((k-2)T_1) = md^{(k-1)}$$

This allows us to generate the following equation based on available data whose solution will provide a least square fit form:

$$\left[d^{(k)}\ d^{(k-1)}\ ...d^{(3)}\right] = m\left[d^{(k-1)}\ d^{(k-2)}\ ...d^{(2)}\right] \tag{14}$$

in which case

$$m = \left[d^{(k)}\ d^{(k-1)}\ ...d^{(3)}\right]\left[d^{(k-1)}\ d^{(k-2)}\ ...d^{(2)}\right]^{-R} \tag{15}$$

and α can be computed by

$$\alpha = \left[(m^2 - m)\ (m^3 - m^2)\ ...\ (m^n - m^{n-1})\right]^{-L}\left[d^{(2)}\ d^{(3)}\ ...d^{(n)}\right] \tag{16}$$

where superscript $-R$ and $-L$ represent the Moore-Penrose pseudo right and left inverse, respectively. Having m and α computed as above we obtain $\lambda = \frac{1}{T_1}(ln(\alpha) - ln(m))$ and therefore $\dot{r}(kT_1) \approx -\lambda\alpha e^{-\lambda T_1} = -\lambda m$.

Inherent in the above is a single exponential approximation to obtain a reasonable estimate of the velocity data. Now we must redo the above development into the proper matrix format. Using data available and the approximated $\dot{r}$ we compute the difference for a specific period of time as $D^i = [\ r(i)\ \dot{r}(i)]^T - [\ r(i-1)\ \dot{r}(i-1)]^T$. It can be shown that $D^i \ = \ M\ D^{i-1}$, where $M \ = \ e^{A\ T}$,

T being the time increment between two consecutive measurements of data, and A the A-matrix of equation 11. Therefore, we can compute M by

$$R_1 \ = \ M \ R_2 \ \Longrightarrow \ M \ = \ R_1 \ R_2{}^{-R} \tag{17}$$

where $R_1 \ = \ [\ D^n \ D^{n-1} \ D^{n-2} \ ... \ D^4 \ D^3 \ ]$; $R_2 \ = \ [\ D^{n-1} \ D^{n-2} \ D^{n-3} \ \ldots \ D^3 \ D^2 \ ]$.

The Spectrum Mapping Theorem [6] shows that the eigenvalues of M, say $\lambda_1{}^M$ and $\lambda_2{}^M$, have the following relation with the eigenvalues of matrix A: $\lambda_1{}^M = e^{\lambda_1 T}$ and $\lambda_2{}^M = e^{\lambda_2 T}$. Therefore $\lambda_1 = \frac{1}{T} ln(\lambda_1{}^M)$ and $\lambda_2 = \frac{1}{T} ln(\lambda_2{}^M)$. The eigenvalues are the roots of the characteristic polynomial of A which is

$$\Pi_A(\lambda) \ = \ det[\lambda I \ - \ A] \ = \ det \begin{bmatrix} \lambda & -1 \\ \frac{\zeta \ \hat{w}_f}{s_c} & \lambda + \frac{\xi}{\hat{\gamma} \ s_c} \end{bmatrix} \ = \ \lambda^2 \ + \ \frac{\xi}{\hat{\gamma} s_c}\lambda \ + \ \frac{\zeta \hat{w}_f}{s_c} \tag{18}$$

Since ξ and ζ are the only unknows at this time, we can compute them by matching the roots of $\Pi_A(\lambda)$ to λ_1 and λ_2 computed above.

An initial estimate of r(0) can also be computed towards the use of matrix M obtained from the observed data. Let

$$P \ = \ \begin{bmatrix} M^2 - M \\ M^3 - M^2 \\ \vdots \\ M^n - M^{n-1} \end{bmatrix} \ \ and \ \ Z \ = \ \begin{bmatrix} D^2 \\ D^3 \\ \vdots \\ D^n \end{bmatrix} \ \ and \ \ x_0 \ = \ \begin{bmatrix} r(0) \\ \dot{r}(0) \end{bmatrix}$$

We know that $Z = Px_0$ and we can compute $x_0 = P^{-L}D$. However, this results in a high initial velocity and penalizes the computation of $r(0)$. The problem is solved by applying a weighted least squares approach [19]:

$$Z \ = \ P \ W \ x_0 \ \ \Rightarrow \ \ x_0 \ = \ ((PW)^T(PW))^{-1}(PW)^T Z \tag{19}$$

where $W \ = \ \begin{bmatrix} w_{r_0} & 0 \\ 0 & w_{v_0} \end{bmatrix}$ is the weight matrix. The weights w_{r_0} and w_{v_0} are usually defined as $\frac{1}{\sigma}$ [19] where σ is the standard deviation computed from the observed data for r and from the estimates of $\dot{r}$. In the case of the STP, due to the exponential decay σ will increase as more data become available. This will make the value of the weights decrease when the opposite behavior is expected. To avoid this problem we defined the weights as $w_{r_0} = \frac{w}{\sigma_r}$ and $w_{v_0} = \frac{w}{\sigma_{\dot{r}}}$ for $w = 1 + e^{-\frac{d/2}{\varrho}}$, where d is the expected deadline and ϱ is the number of observed values used in the computation.

6.3 Software complexity

Software complexity (s_c) has a significant impact on the behavior of our model. In this section we present a way to define a combination of metrics to represent software complexity. Although there is no requirement to use a specific software complexity metric, we choose some existing metrics to exemplify the use of convex combination [20]. This combination offers an opportunity to simultaneously use multiple metrics for software complexity.

Figure 9 lists six different metrics to compute software complexity and shows how these can be combined to estimate the value of the parameter s_c. Two of these six metrics are based on structural properties of programs, three on program size, and one is a combination of the other five. A software complexity metric based on information flow (metric # 1) is defined by Henry and Kamura [14] to capture the relation between procedure size and information flowing into (fan-in) and out (fan-out) of procedures. A survey of metrics based on architectural features (metric # 2) are described by Troy and Zweben [30]. These metrics are based on how internal modules are divided (modularity), how they exchange information (coupling) and how functionally independent they are (cohesion).

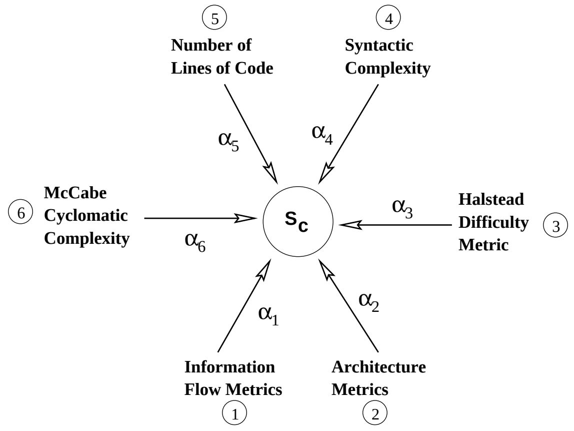

Figure 9: s_c as a convex combination of six software complexity metrics. α_i's are the corresponding weights.

Syntactic Complexity (metric # 4) is defined by Basili [2] based on the attributes product size, control paths, and product decomposition. These attributes are combined to produce the definition of a family of control structure based complexity metrics. Halstead [12] (metric # 3) defined components of software science and program difficulty (D) as a measure of software complexity. The Cyclomatic Complexity (metric # 6) defined by McCabe [23] is based on the number of regions contained in the directed graph of the program. For a large program an average of the Cyclomatic Complexity per method can be used. The number of lines of code (metric # 5) is defined directly and simply as a size relation proportional to complexity.

Based on the six metrics mentioned above, software complexity (s_c) is defined as a convex

Table 3: Quality factors used in the computation of γ.

Quality Feature	Weight Factor (α)	**Range**
Deadline Pressure	0.125	0 to 1
Work force Experience and Expertise	0.250	0 to 1
Test Strategy/Adequacy	0.125	0 to 1
Tool Use/Adequacy	0.250	0 to 1
Test Plan	0.125	0 to 1
Coverage Criteria	0.125	0 to 1

combination

$$s_c = \sum_{i=1}^{n} \alpha_i \; M_i \tag{20}$$

where M_i is a normalized software complexity metric and $\sum_{i=1}^{n} \alpha_i \; = \; 1$.

The software complexity ranges from a lower bound of 0 to an upper bound of ϕ. Parameter calibration techniques [21] can be used to define the upper bound ϕ for individual organizations.

6.4 Quality of the test phase (γ)

We are not aware of any validated and/or widely acceptable metric to measure the quality of the test phase. We assume that γ can be estimated within an organization based on past data and experience. We believe that an organization with a process maturity level of 4 or above on the CMM scale is more likely to be able to estimate γ than one at lower levels of process maturity.

Based on the convex combination approach presented earlier, we provide a guideline to obtain an initial estimate of γ. As in the estimation of s_c, a project manager may freely change values of α_i (restricted to $\sum_{i=1}^{n} \alpha_i = 1$) based upon experience and knowledge about the company and the project. For example, a project manager can determine if coverage criteria is more important than the test plan for a specific project and change the α_i's appropriately. Also, a new quality feature can be inserted or one that is already in use can be removed.

Table 3 lists features to be considered when determining the overall quality of the test phase (γ). The weight and ranges associated with each feature is also listed. The values provided in Table 3 are a starting point for a project manager, and new features can be inserted or removed and the α_i's changed according to company or project characteristics.

7 Case studies

Two case studies were carried out to investigate the performance of the state-based model of the STP. Case study 1, referred to hereafter as CS 1, used the data reported by Knuth [18]. Case study 2, hereafter referred to as CS 2, used data from an ongoing commercial effort to transform a program written in COBOL into a functionally equivalent program in SAP/R3. There is no information available in the open literature on this commercial effort and hence a brief description of the project appears later in this section.

Prior to embarking on a description of the two case studies, we point out a subtle difference between the behavior of $r(t)$ as exhibited by our model and what is likely to be observed in a realistic test process. This difference is illustrated by Figure 10. This figure shows a typical plot of the decrease in errors in a program during the testing and debugging phase [27]. The glaring discontinuities are due to the introduction of one or more new errors during the error removal process. As in Eqn. 6, we assume that the number of errors introduced during the test and debug cycle is less than the number fixed. Solving the model represented by Eqn. 9 and 10 for $r(t)$ does not produce the discontinuities as in Figure 10. Instead, as also shown in Figure 10, our model produces a smooth $r(t)$ whose values can be interpreted as averages of least squares estimates of the actual process.

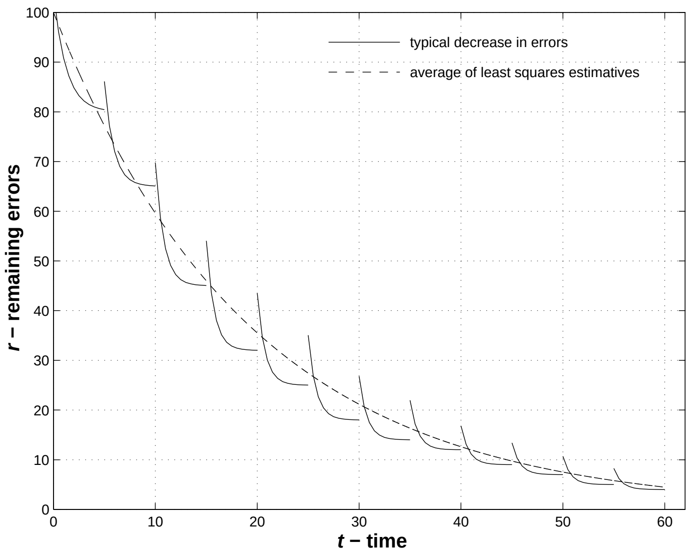

Figure 10: A possible variation in the number of remaining errors (r) over time. Note that the decrease in r s not monotonic though the average least squares estimates are decreasing monotonically.

7.1 Case Study I: TEX78

The test phase for TEX78 lasted 20 days. During this phase a total of 237 errors were found and removed [18]. Having removed these errors, Knuth began using TEX to type Volume 2 of *The Art of Computer Programming* and later to produce the TEX manual. About six months after the beginning of the test phase, other users began using TEX. Considering this information about the testing of TEX we divide the test/maintenance life cycle of TEX78 into three main periods: (i) the initial test phase, (ii) the use of TEX78 by Knuth and a small group of users, and (iii) the release of the product to other users.

Knuth describes 15 types of errors [18]. Nine of these are considered "true errors" and six as enhancements. We do not distinguish amongst errors and enhancements and assume that all errors were detected during the test process.

In Figure 11 we compare the observed rate of decrease of errors for TEX78 (plot 1) and its predicted global approximation (plot 2) using our model. We use "global approximation" here as the approximation generated by a non-switched system, i.e. a system in which the parameters are estimated only once and not in successive periods. We set the software complexity (s_c) of TEX78 as 2.6 based on its size which is 2.6 KLOC. Then, the technique described in Section 6.2 is used to generate the values for the parameters ξ and ζ. For this purpose we set $\gamma = 0.19$ and $w_f = 1.5$. Values of all parameters used in determining the plots in Figure 11 are listed in Table 4.

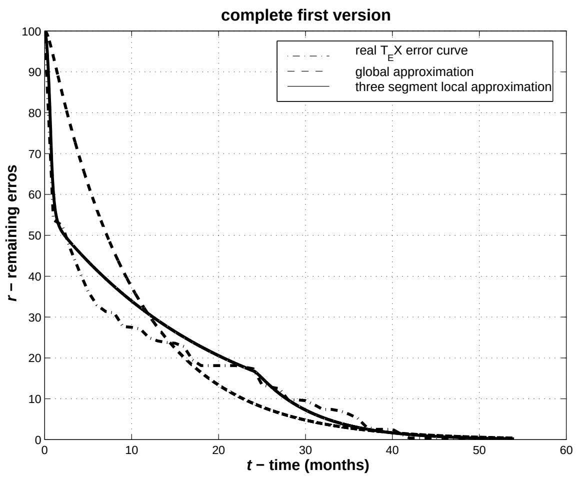

Figure 11: Global and three-segment piecewise approximations of the error reduction in TEX78 .

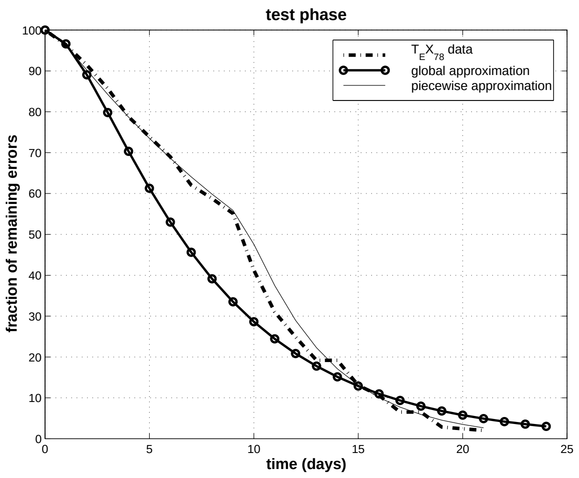

Figure 12: Variation in the number of remaining errors during the test phase of TEX78 .

Table 4: Values of parameters used to obtain the plots shown in Figure 15.

global approximation					
	γ	ξ	ζ	s_c	w_f
	0.19	1.3	0.45	2.6	1.5

Three segment local approximation					
	γ	ξ	ζ	s_c	w_f
Period 1	0.75	1.34	0.22	2.6	1
Period 2	0.42	3.83	0.45	2.6	1
Period 3	0.0027	0.018	0.006	2.6	150

To measure the accuracy of the approximation and to evaluate other approximations, we introduce the integral mean square error between two functions $f_1(t)$ and $f_2(t)$. This error, denoted here by $\hat{\varphi}$, is computed using the following equation:

$$\hat{\varphi} = \sqrt{\int_0^{\infty} |f_1(\tau) \ - \ f_2(\tau)|^2 \ d(\tau)} \qquad (21)$$

Using the above equation we obtain an $\hat{\varphi}_{ga} = 66.49$ between $f_1(\tau)$ that represents the observed data and $f_2(\tau)$ that represents the global approximation. A better approximation results when we use the three test periods defined above. Plot 3 in Figure 11 represents this local three segment piecewise approximation. During the test phase (period 1) we assume that Knuth focused on finding errors and consider the quality of the test phase to be high. This leads to $\gamma = 0.75$ and $w_f = 1$. During period 2, TEX78 was used by Knuth, to type a book and the TEX manual, and by a small group of users. Since neither Knuth nor the users were fully dedicated to finding errors in TEX, we set w_f to a nominal value of 1. We consider that in this period many different features of TEX were used and consequently tested. The quality of the test phase was set to 0.42. During period 3 the quality of the test phase was very low. We consider this phase as an extreme case described in Section 5. For this phase, we set $\gamma = 0.0027$ and $w_f = 150$. The values of the parameters ξ and ζ for the three periods were computed as according to Section 6.2. The values chosen do not represent actual data, since this data are not available. Our point here is that reasonable parameter choices result in a good fit of our model to the data.

Computing the error between the observed data and the local approximation results in $\hat{\varphi}_{la} = 25.59$. This error is much smaller than the error in global approximation and hence favors using piecewise approximation to the modeling of the STP. It also implies that changes in the environment ought to be accounted for and that this could be done by switching to a model with different parameters. The changes in γ and w_f in plot 3 of Figure 11 result from this implication. An even better approximation would be possible if the curve was divided into more segments, but the lack of data about environment changes does not allow such an alternative for TEX78.

Table 5: Parameter values used for computing plots in Figure 12.

global approximation					
	γ	ξ	ζ	s_c	w_f
	0.75	1.34	0.22	2.6	1

two segments local approximation					
	γ	ξ	ζ	s_c	w_f
Period 1	0.6	3.26	0.36	2.6	1
Period 2	0.9	4.29	1.08	2.6	1

We now focus on the initial test phase and predict the output from our model. The predicted values are plotted in Figure 11 and Figure 12. Note that the times are represented in days. The quality factor $\gamma = 0.75$ is computed as a convex combination of two quality factors associated with this period. At the beginning of the test phase there is a learning curve during which we set $\gamma_1 = 0.6$. Then, the test phase performance increases as the need for learning decreases and the hence we set $\gamma_2 = 0.9$. Using these two values we compute an overall quality for the initial test phase to be $\gamma \approx \frac{1}{2}\gamma_1 + \frac{1}{2}\gamma_2 \approx 0.75$. This indicates that for about 1/2 of the duration of the initial test phase the quality of the test phase was 0.6 and 0.9 for the remainder. As shown in Figure 12, the model closely approximates the observed data if the initial period is divided into two parts. This result is supported by the fact that the error resulting from single segment approximation, i.e. global approximation, ($\hat{\varphi}_{t1_1} = 43.92$) is larger than the error that results when two segments are used ($\hat{\varphi}_{t1_2} = 11.35$).

The discrepancy between the estimates resulting from global and local approximations is because the latter approach accounts for forces such as "learning." While using local approximation we account for learning in the test phase by computing the quality of each period of the test phase and then taking a convex combination (weighted average) to obtain a nominal quality factor for the entire period. In addition, the errors found during the test phase do not often contribute to the total debugging effort. A test team believes that a product is mostly error free after the test phase only to find later that this is not true. We interpret this misjudgement as an underestimation of the system complexity.

A total of 239 errors were found by Knuth by the end of the test phase. Suppose that at the start of the test phase one predicted the number of errors to be 239 and so the test team (Knuth) stopped working on the test phase after having found all of the predicted errors. We want to know how good is our model in predicting the initial number of errors ? Using the parameter estimatiuon technique from Section 6.2 we obtained a new initial condition of $r_0 = 481$ errors which represents 93% of the total number of errors (513) found over the life of TEX78. The new initial condition is much more accurate than the earlier prediction of 239 errors that represents only 46% of the total number of errors.

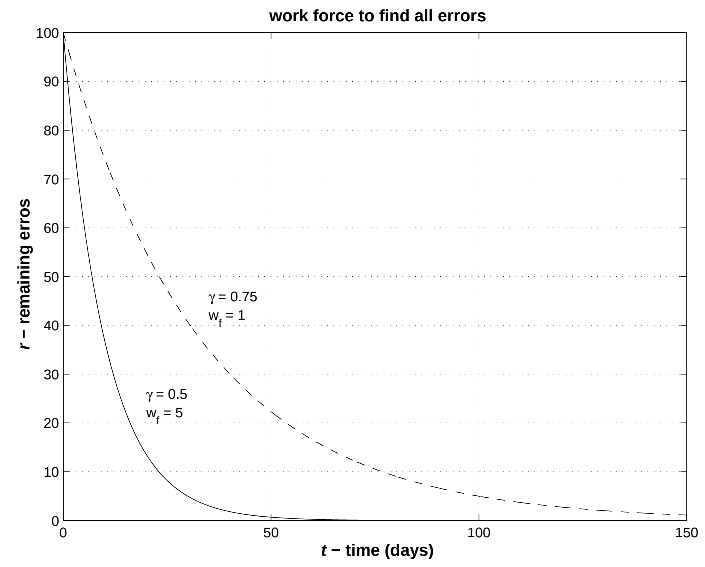

Figure 13: Effect of work force alternatives on the number of remaining errors in TEX78 .

To investigate how the model can be used to predict changes in parameters that affect the rate of error removal, suppose that a good prediction of the total number of errors is available at the beginning of the STP. We ask: "What is the effect of increase in w_f on r?" Figure 13 shows the decreasing error rate to find almost all errors during the test phase for two cases. In the first case (dashed line) the prediction is that it will take about 150 days when $w_f = 1$ and $\gamma = 0.7$. As an increase in the work force often affects the quality of the test phase, for $w_f = 5$ we set $\gamma = 0.5$. The model behavior according to this new pair of values can be observed from Figure 13 where the time spent in testing drops to approximately 50 days. These results are in accordance with the theory developed by Brooks [3]. This theory predicts that increasing the work force by a factor of 5 will not decrease the time to achieve the same error reduction by the same factor. Indeed, under certain circumstances, it will increase the time spent to complete the task. Thus, when good parameter estimates and accurate data are available, the model presented here can be used to optimize the desired results according to restrictions on time and budget.

7.2 Case Study II: The COBOL Transformer Project

7.2.1 Project Description

The case study presented in this section uses data from an ongoing commercial project underway at Razorfish, a company located at Cambridge, MA. Razorfish currently has an application that contains about 4 million lines of code in COBOL. We will refer to this application as S_{COBOL}. This application is to be transformed into a functionally equivalent application in SAP/R3 hereafter referred to as $S_{SAP\ R/3}$. Razorfish is developing a tool, hereafter referred to as *transformer*, to automate this transformation. A general view of the code transformation process and the test strategy used is depicted in Figure 14. The information presented here about the project was obtained through interviews with the project manager, developers, and the test team. Razorfish maintains data on what errors are found in the *transformer*, by whom and when an error was found, and who is responsible for fixing the error. This data was tabulated by the project manager and made available to us.

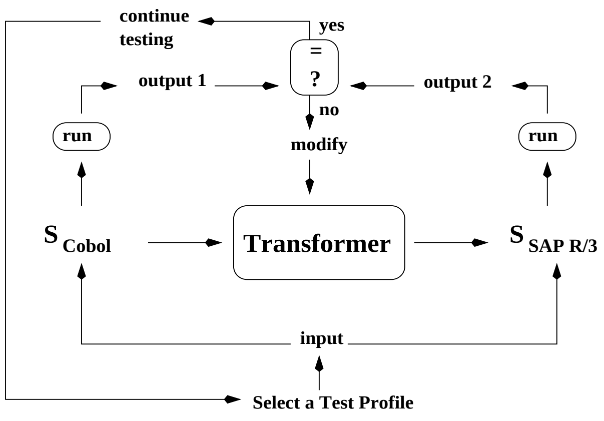

Figure 14: Flow of the test process used in the COBOL Transformer project.

7.2.2 Estimation of model parameters for the COBOL Transformer project

Data from the first 6 weeks of the project was used to obtain an estimate of the initial number of errors. These estimates constitute proprietary data for Razorfish and therefore the values are presented here after normalization. Our estimate of r_0 was considered reasonable by the project manager. This estimate was subsequently improved when data from weeks 6 to 14 became available. By the end of the *transformer* project our estimate was arround 94% accurate. The estimate was computed using the Parameter Estimation technique described on Section 6.2.

Estimation of software complexity

Software complexity (s_c) was computed using the convex combination approach described on Section 6. To measure the complexity of the tool under development we decided to use two size oriented metrics M_1 and M_2. M_1 is the number of ten's of KLOC. The *transformer* has approximately 250,000 lines of code resulting in $M_1 = 25$. M_2 is based on the number of grammar productions used to specify the COBOL syntax. We assume that a sequence of 10 new productions increases the software complexity by one. The *transformer* is designed to deal with different versions and dialects of COBOL. This requirement increased the language specification and complexity significantly. The COBOL specification has around 1,400 productions resulting in $M_2 = 140$. Using the convex combination of M_1 and M_2, and setting $\alpha_1 = 0.8$ and $\alpha_2 = 0.2$ we obtain $s_c = 48$. Here we assume that KLOC account for 80% of the complexity measure.

Estimation of the quality of the test phase

The next parameter to be determined is γ. The testers at Razorfish have significant experience in testing similar systems. The test team also uses a test tool that we believe increases the quality of the test phase. The test tool automates the execution of the native COBOL application and saves the results. It then executes the generated SAP/R3 version and compares the output with that saved earlier. Any discrepancy in the results is reported and prompts a tester to identify the cause of the reported discrepancy and file an error report. This error report is then used by the developers to debug the *transformer* and fix the error found. Upon fixing the errors a regression test is carried out to determine if any new errors have been introduced.

In consultation with the project manager we divided the first 14 weeks of the test phase into three periods. The first period is composed of the first 6 weeks of the test phase. We also decided to use four features to define the overall quality of the test phase (γ). Column 1 of Table 6 lists these four features. Column 2 lists the effective contribution of the feature to the overall quality. Notice that the sum of the α_i's indicates the use of a convex combination. The quality value of each feature will change over the three periods of the project. Each "Period" in Table 6 has two columns. The first column lists the quality level q_{ij} (0 to 1) of the feature i for period j. The second column is the product $q_{ij}\alpha_i$, $i = 1, \ldots, 4$ and $j = 1, 2, 3$. Thus the average quality for period j is $\sum_{i=1}^{4} q_{ij}\alpha_i = \gamma_j$. The details of the choices are beyond the scope of this paper. It should be clear that the features are to be defined on a per project/company basis.

Table 6: Parameters values used in Figure 15.

Quality Features	α_i	Period 1		Period 2		Period 3	
		q_{i1}	$q_{i1}\alpha_i$	q_{i2}	$q_{i2}\alpha_i$	q_{i3}	$q_{i3}\alpha_i$
(1) Experience/expertise of w_f	0.3	0.60	0.18	0.70	0.21	0.80	0.24
(2) Tool use and adequacy	0.2	0.20	0.04	0.50	0.10	0.60	0.12
(3) Test Plan adequacy	0.2	0.80	0.16	0.80	0.16	0.80	0.16
(4) Test cases quality	0.3	0.20	0.06	0.30	0.09	0.77	0.23
		$\gamma_1 = 0.44$		$\gamma_2 = 0.56$		$\gamma_3 = 0.75$	

For period 1 we compute $\gamma = 0.44$. This value for γ is due to the fact that the tool was not as useful in this period as it was in later periods of the project. This was because the testers were testing screen conversions and the generated layout could not be checked automatically by the tool. Also, the testers were using an *in vitro* data base to test the *transformer* and hence the quality of the test cases was not satisfactory.

The second period, weeks 6 to 10, showed improvement due to the use of an improved test set to test the COBOL application. During this period the testers began using real data from a “small” company that makes use of the native COBOL application. This led to an increase in the number of parts of the application that were exercised. An increase in the use of the tool also occurred. Thus, for this second period we compute $\gamma = 0.56$. The third period corresponds to weeks 10 to 14 and was demarcated from the previous phase by the fact that test data from a “large” company using the native COBOL application became available and the system could be exercised more completely. As in the second period, utilization of the tool increased. Thus, for the third period, we compute $\gamma = 0.75$. The values for the test plan adequacy are the same for all periods since it was followed and seemed to be quite appropriate for the *transformer* project. The values for work force experience/expertise increased from 0.6 to 0.8 as the test team adapt to the project.

7.2.3 Size of the work force

The size of the test team remained constant at 3 testers for the period under consideration. This led us to set w_f to 3.

7.2.4 Constant of Proportionality ξ and ζ

The values of the constants of proportionality ξ and ζ were computed for the three periods described early. The values were computed using the Parameter Estimation technique described on Section 6.2 and the results are present in Table 7.

7.2.5 Disturbance force

In Figure 15 we can see the initial expected behavior for the *transformer* project plotted using the project manager expectations. We can also see that the observed behavior diverges from the expected one for the first 14 weeks of the project. This divergence is due to disturbances present during the STP and alternatives to correct its effect, using feedback, is discussed later.

As explained earlier in Section 4.4 disturbance is a force opposing the effective test effort (e_f). We are concerned here with the quantification of the disturbance and not with the identification of its source(s). The disturbance is computed by taking the expected behavior and introducing an opposite force (F_d) that produces the observed behavior, i.e. matches the collected data. F_d is the input in our model and so can be computed by the Eqn. 22 below [6]:

$$F_d(q) \quad = \quad B^T\Phi^T(t_1 - q)K^{-1}(t_0, t_1)\left[x_1 - \Phi(t_1 - t_0)x_0\right] \tag{22}$$

where $\Phi(t_1, t_0) = e^{A(t_1 - t_0)}$ is the state transition matrix and $K(t_0, t_1)$ is the the controllability Gramian [6]:

$$K(t_0, t_1) \quad = \quad \int_{t_0}^{t_1} \Phi(t_1 - q)B\ B^T\Phi^T(t_1 - q)dq \tag{23}$$

The average disturbance for the three periods of the *transformer* project is, respectively, 62%, 36% and 17% of the e_f. This means, for example, that for the first period an opposite force equivalent to 62% of the effective test effort (e_f) was present.

The disturbance is high during the first period and decreases subsequently. The disturbance is due to communication, adaptation, hardware and software failure, illness and other forces not accounted for in our model. Disturbances are always present in the software process and so a disturbance force equivalente to 25% of e_f was extrapolated for the remaining period. Although the disturbance seems to be high, primarily at the beginning of the process, it is usual to have a 40% disturbance under normal conditions as has been pointed out [1]. The high disturbance during period 1 is due to a temporary slow down in the test process that resulted from the discovery of an error. During periods 2 and 3 the test team was more focused and able to concentrate on testing rather than on the source of the error and its removal.

Under certain circumstances the disturbance can increase or decrease due to motivation and scheduled pressure [1]. Considering that the test team at Razorfish has a experience with similar projects, we assume the disturbance to decrease as project proceeds and we set an average of 25% of disturbance for the remaining weeks.

7.2.6 Results

We used our model to understand the behavior of the test process during the first 14 weeks of the COBOL Transformer project. The results were used to predict the behavior for the remaining weeks.

Figure 15 depicts the test phase results of the *transformer* project. The integral mean square error, computed using Eqn. 21, which produced a 2.14 error norm when data from the real project is compared to the model approximation for the first 14 weeks. Parameter values used to generate the data in Figure 15 are listed in Table 7. These parameters and the disturbance inserted during the process produce the approximation depicted in Figure 15.

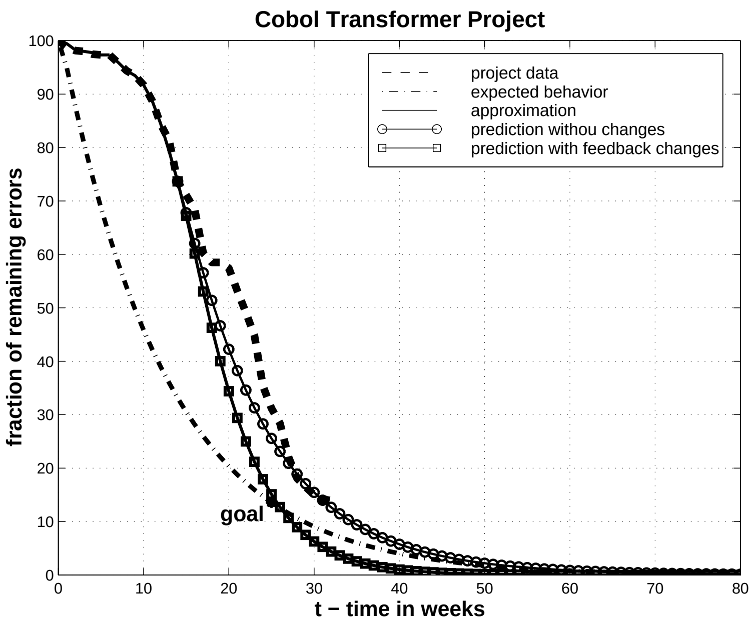

Figure 15: Actual and predicted behavior of the COBOL Transformer project observed in the change in the fraction of remaining errors.

Table 7: Parameters values used in Figure 15

three segments local approximation							
	weeks	γ	ξ	ζ	s_c	w_f	F_d
Period 1	1 to 6	0.44	19.57	0.25	48	3	62%
Period 2	6 to 10	0.56	22.73	0.54	48	3	36%
Period 3	10 to 14	0.75	18.86	0.75	48	3	17%
Remaining Time	14 to ...	0.75	18.86	0.75	48	3	25%

where F_d is the average disturbance in the period

When analyzing the results shown in Figure 15 two assumptions could be made. First, any error in the *transformer* will produce an error in the $S_{SAP\ R/3}$ generated code. Second, not all errors in the *transformer* will affect this specific project, i.e., the transformation from S_{COBOL} to $S_{SAP\ R/3}$ will not be able to exercise all features of the *transformer*. If the first assumption is valid, then according to Figure 15 it will not be possible to accomplish the predetermined deadline. Assuming no change in the test process, i.e. maintaining the same parameters as in the third period described before and keeping a disturbance at 25%, our model predicts that it will take more than 50 weeks to deliver a product with a reasonable level of errors.

The second assumption indicates that by the end of the test phase, i.e. after 25 weeks, some errors will remain in the *transformer* but the goal of the project would be achieved. Stated differently, the generated system will be functionally equivalent to the original COBOL system ensuring a successful project. The remaining errors in the *transformer* can not be found by testing a specific

COBOL system and more effort must be spent to decrease the number of remaining errors to a reasonable level. Thus, the second assumption seems more reasonable than the first one.

It should be clear that the predictions from our model do not depend on any of the above two assumptions. For the purpose of analysis, we are concerned with the remaining errors in the *Transformer* and not in the generated code.

Based on the solution to our model one predicts that it will not be possible to finish the project by the expected deadline. This becomes evident by comparing the approximation to r with the expected curve. Hence we ask: "What changes can be made to the STP in order to meet the deadline?" The use of feedback helps us answer this question.

We note that by week 14 the number of errors dropped to around 67% of their initial value and, if the process continues without any alterations, then it will take around 31 weeks to reach the expected level of error reduction (approximately 14%). Indeed, no adjustments were made in the project and 32 weeks were passed when the project reached the desired level of errors. This result show a 3.2% accuracy in our prediction.

Now, suppose the project manager desires to achieve the same results in only 10 weeks. What "feedback" modifications are necessary?

The largest eigenvalue of a system determines the slowest rate of convergence and dominates how fast the variables converges. Therefore, we need to adjust the largest eigenvalue of the model so that the responses converge to the desired values within the remaining weeks.

Equation 24 below can be used to achieve this goal.

$$r(T+\Delta t) \quad = \quad r(T)\ e^{\lambda_{max}\ \Delta t} \tag{24}$$

where r(T) is the number of remaining errors at timer T, $r(T+\Delta t)$ is the desired value for r after a lapse of Δt time has occurred, and λ_{max} is the eigenvalue to be computed. In the Razorfish project we want the system to converge from 67% at week 14 (r(14)=67%) to 14% at end of week 24 (r(14+Δt)=14%) for $\Delta t = 10$. Solving equation 24 for these values results in $\lambda_{max} = -0.1566$.

The eigenvalues of a system are defined by the roots of the characteristic polynomial ($\Pi_A(\lambda) = det[\lambda I \ - \ A]$). Computing the characteristic polynomial of our model produces

$$det[\lambda I \ - \ A] \quad = \quad det \begin{bmatrix} \lambda & -1 \\ \frac{\zeta\ \hat{w_f}}{s_c} & \lambda + \frac{\xi}{\hat{\gamma}\ s_c} \end{bmatrix} \quad = \quad \lambda^2 \ + \ \frac{\xi}{\hat{\gamma} s_c}\lambda \ + \ \frac{\zeta \hat{w_f}}{s_c} \tag{25}$$

where $\hat{\gamma} = \gamma + \Delta_\gamma$ and $\hat{w_f} = w_f + \Delta_{w_f}$.

To set the eigenvalue of the model described by Eqn. 9 and 10 to -0.1566 we have to make changes in the values of these parameters. Considering that no changes can be done in ξ, ζ and s_c, we are left with two options: increase the work force ($\Delta_{w_f} \ > \ 0$) or improve the quality of the test phase ($0 \ < \ \Delta_\gamma \ < 0.25$).

Varying Δ_{w_f}, keeping all other values constant and then finding the roots of the characteristic polynomial produces the results depicted in Figure 16(a). We can observe that λ_{max} reaches the desired value of -0.1566 when Δ_{w_f} reaches 0.7. This implies that the w_f has to be increased by 0.7 in order to accomplish the deadline, assuming all other parameters are kept constant. The feedback result of increasing the work force by 0.7 is presented in Figure 15. Figure 16(b) presents similar results achieved from the variation of Δ_γ. As can be observed, an increase of 0.11 in γ is needed to achieve the desired results.

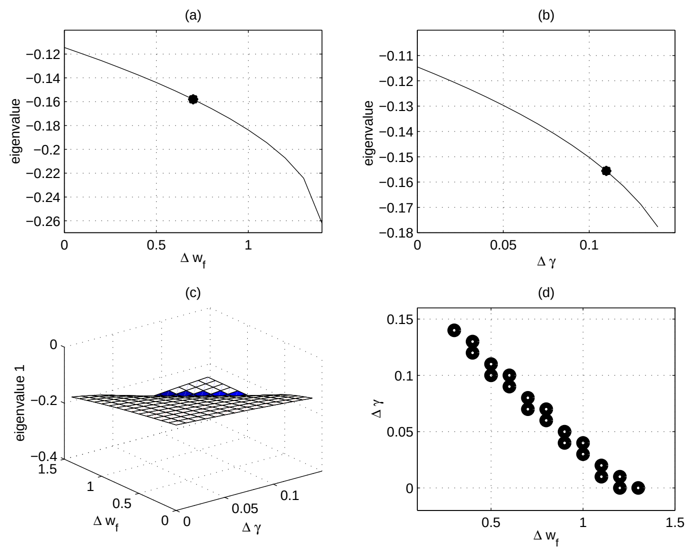

Figure 16: Relationship that must be maintained between the largest eigenvalue of the system, Δ_γ, and Δ_{w_f} to achieve the desired number of remaining errors.

Figure 16(c) presents the combined results when Δ_γ ranges from 0 to 0.15 and Δ_{w_f} ranges from 0 to 1.5. A project manager can use these results when more alternatives are available, such as a deadline extension. In the *transformer* project we are interested in finishing the test phase within the predetermined deadline and all possible combinations of increasing w_f and/or γ to accomplish this task are depicted in Figure 16(d).

The model can also be used to analyze different alternatives according to flexibility of the deadline and resource availability. That is, if 0.7 people are not available, the manager can choose the alternatives of how many testers can be inserted and how much the deadline can be extended. It is basically an exercise of maximizing customer satisfiability according to resource limitations. As stated before, optimization techniques are available in Control Theory [9] to provide these results, but it is beyond the scope of this paper.

In general we can conclude that our model behavior is reasonably accurate when applied to the TEX78 error log and the COBOL *Transformer* project and that feedback can be used to answer questions related to performance and cost of the STP.

8 Related work

An overview and analysis of four different approaches to modeling the software process is presented in this section. Our analysis is based on features defined in Section 8.1. The modeling approaches selected for analysis are the ones most the relevance to our work. Several other approaches to modeling the SDP, not reviewed here, are found in the literature. A table listing the characteristics of the four approaches and a comparison with the classical state variable based approach is also presented. Though our analysis is concerned with the entire SDP, most of the features analyzed here can be also considered for the STP itself.

8.1 Characteristics of approaches to modeling the SDP

In this sub-section we describe and compare four approaches to modeling the SDP. The comparison is based on model characteristics that we believe are most relevant to any approach to modeling the SDP. These characteristics are described below.

1. *Dependency on life cycle model*: Some organizations use life cycle models available in the open literature while others develop their own models. Thus, a software process modeling approach that is applicable in a variety of life cycle paradigms is needed. The dependency of the modeling approach on the life cycle model used is intended to measure how much an approach is committed to a specific life cycle.

2. *Optimization*: Though software organizations continue to attempt to establish a stable and predictable SDP environment, optimization is the next natural step towards a reduction in production costs and improvement in productivity. Optimization aids in the determination of how simplicity and effectiveness of a modeling approach are related to the use of optimization methods available to the specific approach.

3. *Self regulation*: When an expected behavior is available and is accurate enough to guide the current behavior of a process, a self regulation mechanism can be used to show how this goal can, if possible, be achieved. The self regulation will present the effects of an overshoot related to cost and schedule issues. This feature will characterize the availability of a self regulation mechanism on the approaches.

4. em Coupling and cohesion: The coupling and cohesion determines how an approach handles various phases of the SDP. Cohesion indicates how independent is the model for one phase from the model for other phases. Coupling measures the interference of parameters/data within a phase with the behavior of other phases. A high cohesion and a low coupling are desired.

5. *Completeness*: Completeness measures how much of the entire SDP is modeled by the approach.

6. *Caliberation*: Calibration measures whether or not an approach includes feasible methods calibrating the model. Although calibration can be done by "observation", the presence of a methodology to guide the user in calibrating the model according to an organization's needs is highly desirable.

7. *Friendliness*: Friendliness measures the ease of applicability of an approach to an organization or a project. In this context, some relevant questions are: Can the approach be applied by a manager having a minimal knowledge of the approach ? Is only a knowledge of the software process enough ? Friendliness also accounts for the amount of effort that ought to be expended in the acquisition of minimal knowledge about a specific approach.

8.2 Software Project Dynamics

The work by Adbel-Hamid and Madnick [1] makes two major contributions to the modeling of SDP. The first contribution is in that the the model is integrated and hence provides a macro understanding of the SDP through the integration of micro components. The second contribution stems indirectly from the first one. It is in that the model can predict the general behavior of the SDP by propagating the effect of changes from one phase to the subsequent phases. This ability to predict enhances our understanding of how a local change will affect the behavior of the entire project. The model's suitability to simulation is another useful characteristics of their work. Next, we provide a brief description of the model.

8.2.1 The Software Development Process

The model developed by Abdel-Hamid and Madnick is divided into four subsystems. The subsystems and the connections amongst them are depicted in Figure 17 . From this figure it is easy to see that the subsystems are affected by each other. The subsystems are described next.

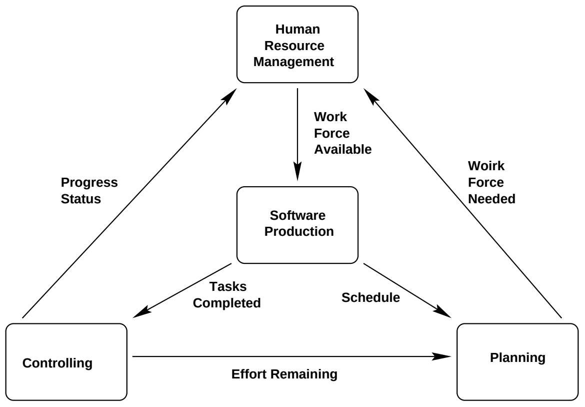

Figure 17: Software development subsystems used in the Systems Dynamic Approach of Abdel-Hamid and Madnick [1].

Human Resource Management: This subsystem models the control alternatives of how and when the work force should be hired or transferred and trained. The effects of these actions propagate to the connected subsystems. Two levels of work force are defined: "newly hired" and "experienced". Based on the number of people available from each level and the training and adaptation time necessary for each one, the Human Resource Management subsystem determines the nominal work force available.

Software Production: This subsystem is too complex to be considered as a single unit, and hence is divided into four interconnected sub-subsystems named sectors. The Manpower Allocation sector takes the total work force available as a primary input. It accounts for parameters such as deadline pressure, work to be done, training overhead, and the desired quality to allocate the work force to quality assurance, rework, software development and testing. The second sector is Software Development that is based on two primary components: productivity and tasks. That is, based on the productivity of the work force available to development and the number of tasks at hand, the model can define a productivity rate and determine the time required to complete the tasks. Quality Assurance and Rework is the third sector. It manages the generation of errors, their detection and correction during software development. The last sector is System Testing that models the behavior of the error detection process. The errors generated and undetected in the previous phases must be detected in this sector before releasing the product. An error is not a static parameter and is assumed to propagate from one phase. The model is designed to account the propagation and multiplication factors to predict the effort needed to detect and correct the errors remaining.

Control: The ability to control a process is dependent on the ability to measure its progress. This is not easy accomplish as software development is not a quantifiable task. Progress is measured in the model using two parameters: rate of expenditure of resources and percentage of accomplished tasks. The first one is used during the early phases of software development and its weight decreases with the approach of the final phases. Adjustments are needed according to progress or according to the amount of underestimation. If the project is behind schedule the development team needs to decide how much work they can absorb by "working harder" and then the model parameters can be adjusted to absorb the extra work force and its effects. A similar behavior is presented when the project is ahead of schedule. Underestimation of the project's size also leads to adjustments, i.e., more tasks need more effort and/or more time for completion of the project.

Planning: The primary goal of the Planning subsystem is to take initial estimates at the start of the project and review these as the project continues. Then, in accordance with the work force needed, the project manager can decide to extend the deadline, hire more people, or increase the work force level.

8.2.2 Model analysis

The sequence of software development phases assumed in Abdel-Hamid's model suggests the use of a waterfall life cycle model. We therefore consider the model to be highly dependent on which model of life cycle is used. The model is not restricted to any specific methodology. The strategy of modeling the design and coding phases as one decreases model cohesion and increases coupling. Therefore a high coupling and medium cohesion was determined for the model. Since the model does not address all phases of the SDP a medium completeness was defined. The optimization issues are related to simulation and will have a medium level as Software Process Simulation described on Sub-section 8.4. The self regulation and calibration features are not addressed in their work and so characterized as low. Finally, the model appears to be relatively easy to use and having a medium level of friendliness.

8.3 Statistical Process Control

Statistical Process Control, as described by Florac and Carleton [8], is not a model in itself. However, it provides useful tools to improve the controllability of the SDP and hence an understanding of its concepts is important to our research. Two concepts used in Statistical Process Control most relevant to our work.

- Stability - a process is under control, or in a stable condition, if its predicted behavior is within limited to expected variations.
- Capability - a process under control is capable if under the conditions of stability it will be capable of accomplishing the desired results.

A Control Chart is the most common tool used to analyze whether or not a process is under statistical control. There are many different types of Control Charts such as X-Bar, Range Charts, U Charts and Z Charts. Each type of Control Chart is supposed to be used according to specific conditions related to the data available. We will exemplify the use of a Control Chart by describing a X-Bar chart.

X-Bar charts are based on averages and applied when data is grouped in subgroups of size 2 or more. Three lines are presented in a X-Bar chart: (CL) center line ; (UCL) upper control limit ; and (LCL) lower control limit. These lines are computed based on observations of the running process and they represent the observed behavior of the process, not the desired one. The center line is the average of averaging each subgroup. The upper and lower control limits are computed by adding and subtracting 3σ from the center line, where σ is the estimated standard deviation. An example of a X-Bar chart is depicted in Figure 18. The X-Bar chart represents the average daily hours per week and it is used to observe if the development effort is according to the predicted one (40 staff hours per day). The example is the same presented by Florac and Carleton [8].

A process out of control if one of the following tests fail [8].

- Test 1 : a single point is outside the limits set by LCL and UPL.
- Test 2 : at least two out of three successive values fall on the same side of, and more than 2σ units away from, the center line.
- Test 3 : at least four out of five successive values fall on the same side of, and more than 1σ unit away from, the center line.
- Test 4 : at least eight successive points fall on the same side of the center line.

It can be observed in Figure 18 that, according to the tests defined above, the process under observation is not out of control, that is, the process is stable. Although the process is stable,

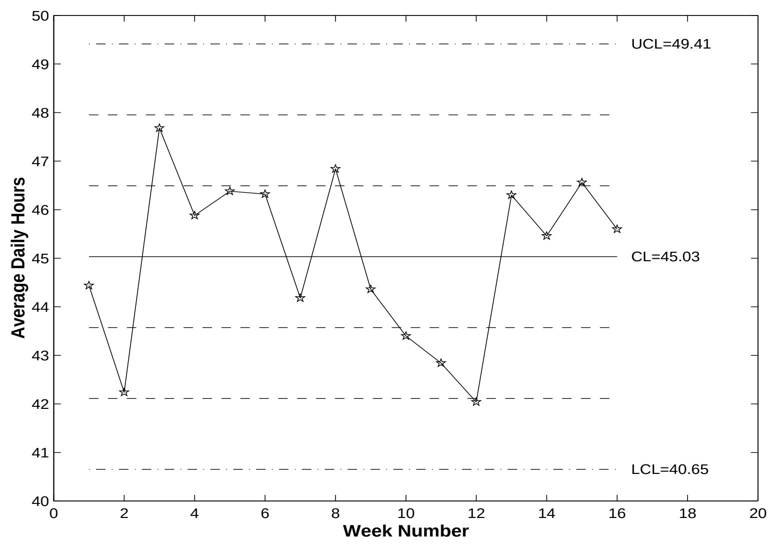

Figure 18: A sample control chart used in statistical process control.

the average daily hour observed is around five hours above the expected effort. Thus, we need to analyze if the process can accomplish the planned task within the predetermined cost and schedule. A Capability Histogram can be used to accomplish this task by determining if a process is outside the expected limits. This fact represents a higher probability of generating non-conforming results and indicates that the process should be adjusted in order to achieve the expected results.

8.3.1 Model Analysis

The applicability of SPC is independent of the life cycle model and the development methodology. It does require that a measurement process be instituted with the SDP. Optimization of process parameters can be addressed in SPC though it requires an analysis of many alternatives. That is, when the number of parameters that affect the SDP is relatively high, the combinations of possible values are even higher and the analysis of all alternative choices becomes difficult if not impractical. The problem is not with the analyses of alternatives, but in their enumeration. SPC does not address self regulation. Cohesion, coupling, completeness, calibration and friendliness are not applicable to SPC.

8.4 Software Process Simulation

We discuss the software process presented by Hansen [13]. This process assumes the waterfall model of the software life cycle. The software to be developed had a total of 100 modules divided into three categories: (i) Type 1 - 20 modules; (ii) Type 2 - 40 modules and half of the time to be completed when compared to Type 1; and (iii) Type 3 - 40 modules and one-fourth of the time to be completed when compared to Type 1. It is assumed that all phases in the waterfall module have dedicated staff. The rework rate and the rate specifications are released to requirements are also defined.

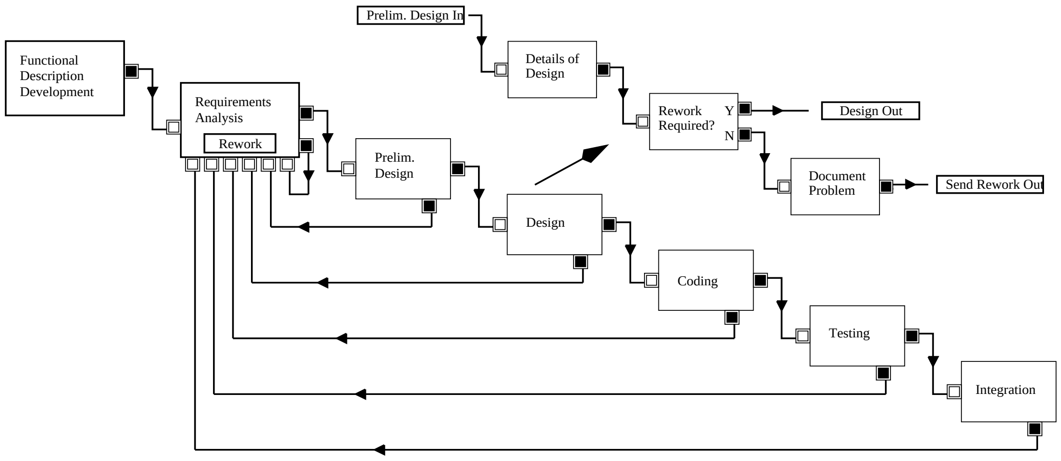

Figure 19: Waterfall model with feedback loops as used in the simulation of the software development process.

Figure 19 shows the waterfall model used where each block is associated with the executable code that will run the simulation. Each block can be divided into sub-blocks and further forming a hierarchical and more detailed representation of the process. Figure 19 also depicts the expanded version of the design phase. The model described by Hansen also provides mechanisms to compute the cost of each phase given the attributes associated to them.

After modeling the entire SDP, simulation is used to examine alternatives to improve or control the SDP. Sample questions of interest are: What happens if Type 1 modules have a higher priority than modules of Type 2 and Type 3? Should this priority be preemptive or not ? What happens if there is a decrease in the rate at which specifications are released ? Simulation provides reasonable answers to these questions. The answers provide enough data to a manager to select the most appropriate alternative(s) that satisfy the imposed constraints.

8.4.1 Model Analysis

Software Process Simulation models can be developed in accordance with the life cycle model and the methodology used. The life cycle is represented by the way and sequence the software phases are considered and features regarding the methodology can be modeled through parameters and variables definitions. Thus both features have a low level, as desired. The same comments regarding optimization issues for Statistical Process Control are valid here. A self-regulation mechanism is not present in this approach.

Since a model can be defined on a per-company basis, it is possible to achieve a high level of cohesion and a low level of coupling thereby making Software Process Simulation models attractive. A new model can be easily defined if changes are detected in the SDP. Restriction regarding the completeness of a model are due to creative aspects of the initial phases of the SDP and present the same measurements and evolution problems as other dynamic approaches. The models can also be caliberated empirically, but, as far as we are concerned, a methodology to do so is not available. The use of simulation requires knowledge about how to model and how to analyze the results obtained. Thus, the simulation approach presents a medium level of friendliness since it appears not to be natural to represent the SDP.

8.5 Object-Oriented Modeling Approach

The observation that "Software processes are software too" [25] makes object-orientation an attractive approach to model the SDP. The maintainability and the abstraction levels provided when an OO approach is used justify its application. All characteristics of object-orientation, such as polymorphism, inheritance and encapsulation can be used to properly model the SDP. DRAGOON, an Ada-like syntax programming language, and Unified Modeling Language (UML) are two examples of the use of object-oriented techniques for such modeling process [15, 28]. Although UML does not account for dynamic behavior, when combined with other techniques it can be a reasonable modeling alternative. Jager et al. describe the application of UML to model the SDP [15].

Using an OO approach, more specifically, the approach described by Riley [28], models for the phases of the SDP can be defined. Teams and Activities are associated with each defined phase. Teams, composed by Roles, perform Activities and Artifacts are produced as a result of that. Each Role is assigned to an employee. Riley presents an object relationship model describing this structure as can be seen in Figure 20.

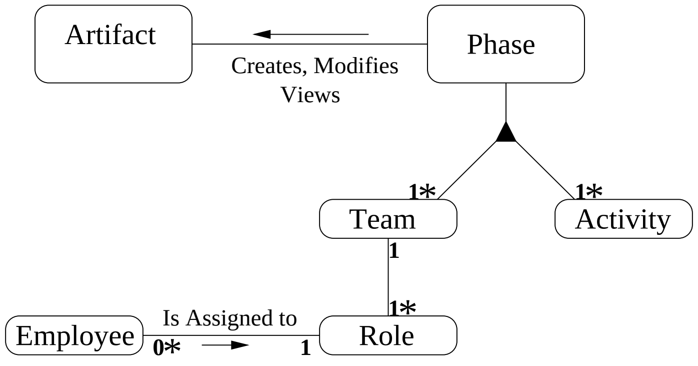

Figure 20: Object relationship model for one phase of the Software Development Process.

Activity flows provide the sequence of realizations and the Artifact_base stores the output of such activities. Many activities can be defined to fully represent the SDP. Besides the development, coding and testing, one can insert quality assurance activities according to project's requirements. A hierarchical structure of some phases of the SDP and their related activities are provided by Riley. This structure shows how natural and appropriate an objected-oriented approach is to model the SDP. The DRAGOON approach also provides ways to collect metrics and to support the concurrent aspects of the SDP.

8.5.1 Model Analysis

An objected-oriented approach can be used to define any life cycle model of the SDP in a natural way. Therefore, the OO approaches presents a low level of life cycle dependency and a high level of friendly. The abstraction level makes OO more appropriate to model the initial phases of the SDP. Thus, completeness has a high level when OO is under consideration. Also, the object-oriented features allows a SDP model that can be easily maintained and so achieving reasonable levels of coupling and cohesion.

It appears that an object-oriented methodology is more appropriate for use when the same approach is used to model the SDP. Although an OO methodology seems to be the natural choice in this case, it does not prevent the use of a different methodology to model the software product. Therefore, we define a medium level of methodology dependency when OO is in use.

Object-oriented languages can be used to simulate the model for the SDP and so this approach has the same optimization, self regulation and calibration levels as Software Process Simulation.

8.6 Classical control theory

The state variable model presented in this paper is a system representation that allows the use of techniques from classical control theory. The interconnected four phase feedback structure of Figure 1 allows for a variety of possible information flows within the SDP. Hence it is sufficiently rich to represent the software development life cycle paradigms found in the literature [27]. It can be adapted to specific organizational structures simply by removing irrelevant feedback paths. For example, remove all paths but the paths connecting two adjacent phases and the feedback path from test to specification in Figure 1. Determine breakpoints in time, that is, how frequently the whole cycle will be re-executed. These changes will result in a spiral life cycle model. Similar changes can be made to fit other life cycles models. Features that depend on a specific development methodology are accounted for model parameters. That is, the theory developed is applicable to any phase within the life-cycle and any software development methodology. Therefore a low level of dependency can be assigned to our approach. The high level of optimization, self-regulation and system calibration are justified in Section 4.

The completeness of this approach can not be addressed yet because models for the other phases of the SDP are not available. Although the initial phases of the SDP present a high level of creativity and will not be easily modeled, according to our experience modeling the STP and comparing to models from other areas presenting similar problems, we believe it will be possible to achieve, at least, a medium level of completeness. That is, we believe it is possible to model the whole SDP but the amount of detail that can be inserted in the model it is not completely clear for us.

The classical control theory approach is not friendly when presented in a state variable representation. Thus, we classify our approach as having a low level of friendliness. Despite that, the low friendliness presented by this approach can be improved by providing a interface hiding the details of the equations and just balancing the forces acting on the SDP.

Table 8: Comparisons

Features	Approaches				
	Software Project Dynamics	Statistical Process Control	Software Process Simulation	Object Oriented Modeling	Classical Control Theory
Life Cycle Dependency	H	L	L	L	L
Optimization	M	M	M	M	H
Methodology Dependency	L	L	L	M	L
Self Regulation	L	L	L	L	H
Coupling	H	NA	L	L	L
Cohesion	M	NA	H	H	H
Completeness	M	NA	H	H	NA
Calibration	L	NA	L	L	H
Friendliness	M	NA	M	H	L

NA-Not Applicable L-Low M-Medium H-High

Table 8 is a summary of our comparison between the modeling approaches described in this paper. Three rating levels namely, LOW, MEDIUM and HIGH, are used to measure how an approach fits the desired feature. A rating of LOW is desirable for cohesion, life cycle and methodology dependency and a rating of HIGH for the remaining features.

As can be observed in Table 8, the Classical Control Theory approach shares the desirable features of the other approaches while providing improvements in optimization, self-regulation and calibration. We believe that ratings for friendliness and completeness of the Classical Control Theory can be improved.

9 Summary and discussion

The widespread use of differential equations and state variable approach to model different types of systems, combined with the advantages of using classical control theory techniques, encouraged us to investigate a formal approach to modeling the STP. Results from two case studies suggest that the formal approach presented in this paper is reasonably accurate in predicting the behavior of the test process. The use of model switching to handle changes in the environment improves flexibility and applicability of the model. Even though our model does not account for several features of the STP, such as adaptation time and communication overhead, we believe that in its present form it is useful in that it captures the essential behavior of the STP. The model can also be used to reduce the cost of STP and improve its performance in the presence of perturbations. The behavior the model for STP enhances our belief that the application of state variable approach is appropriate and likely to result in an improved understanding of the changes during the software process.

A comparison of the the state variable approach to four other different modeling approaches reveals that it shares many of the "good" aspects of other approaches and also presents some advantages over them. These advantages stem from the application of classical control theory and aids in the improvement of STP controllability. This occurs through the use of a self-regulation mechanism. Good model calibration in a per-project/per-company basis and the usefulness optimization are also due to its reliance on classical control theory.

The availability of an analytical model, ability to quantify and estimate model parameters, and an ongoing measurements process are the basic requirements for a successful application of any modeling approach grounded in classical control theory. Organizations at levels 4 and 5 of the Capability Maturity Model (CMM) [26] are likely to have a measurements process in-place. Data collected through this process could be used in the estimation of model parameters. However, the organization level of a company is not a requirement for a successful application of such approaches. Even when the SDP is not well defined, the state variable approach can be applied when measurements of the variables to be controlled and estimates of model parameters are available. Therefore, the model can also be applied to organizations at level 3 or below restricted to measurements availability. That is, even though organizations at levels 1 to 3 does not share the same environmental aspects of levels 4 and 5, they can benefit from the use of the model described here. However, we can not expect the same accuracy as one is likely to achieve within organizations at levels 4 and 5. If the software development process is not very well defined, is is unrealistic to expect availability of accurate data. Despite that, the use of our model might force an improvement in the quality of the data collected and perhaps in the SDP itself.

Several aspects of modeling the STP remain to be investigated. Parameters, such as ζ, need to be defined in a more precisely to include more aspects of the STP. A sensitivity analysis of model parameters is under study and will likely guide us in further refinement of the model. It will show us how changes in parameters affect the model. By comparing these results with expected behavior, we will be able to determine how the model ought to change to accomplish the expected behavior. We believe that success in this research will lead to a process which when implemented rigorously would lead to reduced delays in product development and higher reliability of the product shipped.

9.1 Barriers to the use of our model

Any theory, specially one that is new, is likely to face barriers to its use. The theory of process control, based as in this paper on the theory of feedback control, faces several barriers three of which are identified and discussed below.

Estimation of parameters: Estimation of several parameters is a pre-requisite to the use of our approach. Lack of standardized definitions and widely accepted procedures for estimation make parameter estimation an error-prone task. For example, there is no single definition of software complexity. Thus, as described earlier, one could combine several complexity measures and compute a composite complexity. However, the inclusion of reusable code adds an added dimension to the computation of complexity. The quality of the test phase is a subjective measure. No two test managers are likely to arrive at an identical quantification. We believe that experience with the test process and data from previous test processes within the same company could help in arriving at accurate parameter estimates. A sensitivity analysis of the model is underway and when completed it would help us understand the impact of errors in the estimates of parameters on the predicted values of state and output variables.

Background of test manager: Our model is formal and based on a knowledge of mathematics that very few test managers are likely to possess. Thus, one might argue, how could a test manager use such an approach in practice? We believe that this barrier could be overcome effectively by packaging our approach in a tool. This will hide the details, such as the solution of differential equations, not needed by the test manager. The manager could then interact with the tool by providing parameter estimates and the tool in turn provides estimates of the output and state variables. Thus the tool and the test manager could be partners in the feedback control process.

Process elements: humans versus devices: The theory of automatic control was developed, and is applied, in situations where the various control elements are electromechanical devices and not human beings. In the software test process, the control signals are, among several things, directly effecting people such as when the work force is to be increased or when the quality of the test process is to be increased. Furthermore, the feedback control loop is closed by a human being, namely the test manager. A natural question to ask is: "How effective will a formal control technique be in such a human-intensive environment?" Of course only time will offer an answer to this question. However, we believe that a sound theory of software process control, that has proven to be effective in controlled experiments, is more likely than not to encourage test managers to adapt it.

References

[1] Tarek Abdel-Hamid and Stuart E. Madnick. *Software Project Dynamics: an Integrated Approach.* Prentice Hall Software Series, New Jersey, 1991.

[2] V. R. Basili and D. H. Hutchens. An empirical study of a syntatic complexity family. *IEEE Transactions on Software Engineering*, SE-9(6):664–672, 1983.

[3] Frederick P. Brooks. *The Mythical Man-Month.* Addison Wesley, 1995.

[4] G. Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Transactions on Software Engineering*, 24(11), November 1998.

[5] G. Cugola and C. Ghezzi. Software processes: A retrospective and a path to the future. *Software Process Improvement and Practice*, 1999.

[6] Raymond A. DeCarlo. *Linear systems : a state variable approach with numerical implementation.* Upper Saddle River, New York: Prentice-Hall, 1989.

[7] Willa K. Ehrlich, John P. Stampfel, and Jar R. Wu. Application of software reliability modeling to product quality and test process. In *Proceedings of ICSE*, 1990.

[8] William A. Florac and Anita D. Carleton. *Measuring the Software Process: Statistical Process Control for Software Process Improvement.* SEI Series in Software Engineering. Addison-Wesley, 1999.

[9] Bernard Friedland. *Advanced control system design.* Upper Saddle River: Prentice-Hall, 1996.

[10] Glenn Fulford, Peter Forrester, and Arthur Jone. *Modeling with Differential and Difference Equations.* Cambridge University Press, Cambridge, United Kingdom, 1997.

[11] Carlo Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[13] G. A. Hansen. Simulating software development processes. *IEEE Computer*, January 1996.

[14] S. Henry and D. Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software Practice and Experience*, 14(6):561–573, 1984.

[15] Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using uml for software process modeling. In *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT symposuim on Foundations of software engineering*, pages 91 – 108, 1999.

[16] Karama Kanoun, Mohamed Kaanicke, and Jean-Claude Laprie. Qualitative and quantitative reliability assessment. *IEEE Software*, 14(2):77–87, 1997.

[17] Hassan Khalil. *Nonlinear Systems.* Upper Saddle River New York: Prentice-Hall, 1996.

[18] Donald E. Knuth. The errors of tex. *Software-Practice and Experience*, 19(7):607–85, 1989.

[19] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems.* Society for Industrial and Applied Mathematics (SIAM), 1995.

[20] Steven R. Lay. *Convex Sets and their Applications.* John Wiley & Sons Inc., New Yor, 1982.

[21] Lennart Ljung. *System identification: theory for the user.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[22] David G. Luenberger. *Introduction to Dynamic Systems: theory, models and applications.* John Wiley & Sons, 1979.

[23] T. H. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(6):308–320, 1976.

[24] William Messner and Dawn Tilbury. *Control Tutorials for MATLAB and Simulink: A Web-Based Approach.* Addison-Wesley, 1999.

[25] L. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, 1987.

[26] Marck C. Paulk and et al. Capability maturity model for software. Technical report, Software Engineering Institute, Carnegie Mellon University, 1993.

[27] Roger S. Pressman. *Software Engineering - A Practitioner's Approach.* McGraw-Hill International Editions, 1992.

[28] John D. Riley. An object-oriented approach to software process modeling and definition. In *Proceedings of the 1994 conference on TRI-Ada*, pages 16 – 22, 1994.

[29] Norman F. Schneidewind. Measuring and evaluating maintenance process using reliability, risk, and test metrics. *IEEE Trans. on Software Engineering*, 25(6):769–781, 1999.

[30] D. Troy and S. Zweben. Measuring the quality of structured designs. *The Journal os Systems and Software*, pages 113–120, June 1981.

[31] Alan Wood. Predicting software reliability. *IEEE Computer*, 29(11):69–77, 1996.

Appendix: Linear System Theory: an overview

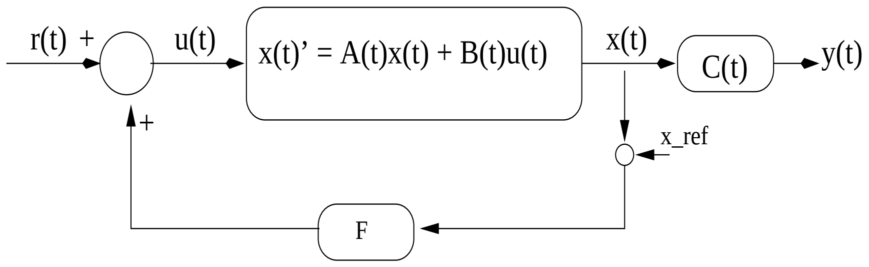

Figure 21: General structure of a linear feedback model.

Linear state feedback models have provided useful representations for large classes of engineering, biological, and social processes [6, 9]. Our purpose is to apply aspects of this modeling and control theory to the software development process with a focus on the testing phase in this paper. Although our intent is use Linear Time Invariant (LTI) state models, we can always generalize our methods to nonlinear models [17] as the need arises.

We are interested in linear feedback systems to enable us to model software development processes wherein feedback can play a crucial role. Figure 21 shows the input-output relationship in a system with feedback. We use the following terminology to represent various items of interest in Figure 21: $x(t) \in R^n$ is the state vector representing the dominant variables characterizing the process; $y(t) \in R^q$ is the output/measurable variables; $r_{ref}(t)$ is a reference input signal; d(t) is a possible disturbance A, B and C are parameter matrices; and F is a feedback matrix designed to achieve the desired systems objectives. Various feedback techniques can be found in the literature [22, 6]. The resulting feedback compensated system is

$$\dot{x}(t) = A\,x(t) + B\,u(t) \tag{26}$$

$$y(t) = C\,x(t) \tag{27}$$

$$u(t) = F\,x(t) + r_{ref}(t) + d(t) \tag{28}$$

Incorporating 28 into 26 produces the feedback compensated dynamics in the following form:

$$\dot{x}(t) = (A + B\,F)x(t) + B\,r_{ref} + B\,d(t) \tag{29}$$

Feedback System: An Example

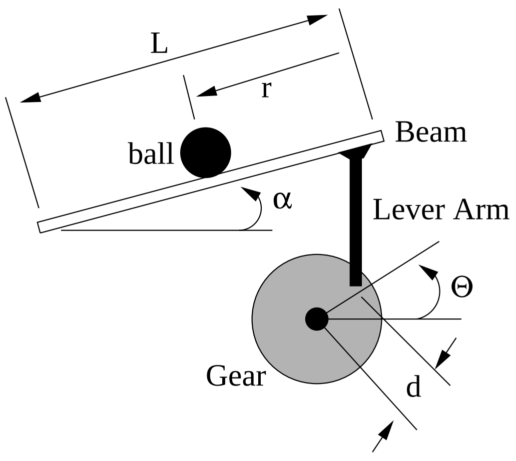

Figure 22: An example of feedback as found in the Ball & Beam problem [24].

A ball and beam system is pictured in Figure 22 having the following linearized differential equation model:

$$(\frac{J}{R^2} + m)\ddot{r} = -mg\alpha \tag{30}$$

$$\alpha = \frac{d}{L}\theta \tag{31}$$

where: m is the mass of the ball, R its radius, d the lever arm offset, g the gravitational acceleration, L the length of the beam, J the ball's moment of inertia, r the ball'sposition coordinate, α is the beam angle coordinate; and θ is the servo gear angle. The gear and lever arm control the angle θ which can be adjusted to make the ball roll and stop at a predetermined position along the beam.

To achieve this goal using state variable control theory, the model of Eqn. (30) and (31) is converted to the following LTI state model:

$$\begin{bmatrix} \dot{r} \\ \ddot{r} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} r \\ \dot{r} \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{mgd}{L(\frac{J}{R^2}+m)} \end{bmatrix} \theta \tag{32}$$

$$\begin{bmatrix} r \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} r \\ \dot{r} \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \theta \tag{33}$$

If we change the angle θ in the open loop model, the ball will start moving and without a regulation mechanism it will fall off the beam. Thus we need a self regulating mechanism to properly position the ball. This can be done by using a state feedback controller

$$\theta = Fx + r_{ref} = \begin{bmatrix} 323.8095 & 19.0476 \end{bmatrix} \begin{bmatrix} r \\ \dot{r} \end{bmatrix} + 0.25$$

which results in a "closed loop system", $\dot{x} = (A + BF)x + Br_{ref}$. The feedback gain matrix F was designed to obtain a settling time of less than three seconds and an overshoot of, at most, 50%. Hence the eigenvalues of $(A + BF)$ were chosen as $-2 \pm 8j$ to obtain these results. The real parts of these eigenvalues determine the settling time and the imaginary part the overshoot. The relationship between the settling time, overshoot and the reference input (r_{ref}) are very well documented in the literature [9, 6] and hence not described here.

The output for the system described above is shown in Figure (23), where the reference input (r_{ref}) is 0.25. The feedback gain matrix is determined by means of the "place" command in MatLab for the eigenvalues defined above. Although the overshoot could be avoided, we set the eigenvalues to produce a large overshoot to stress the self-regulating mechanism.

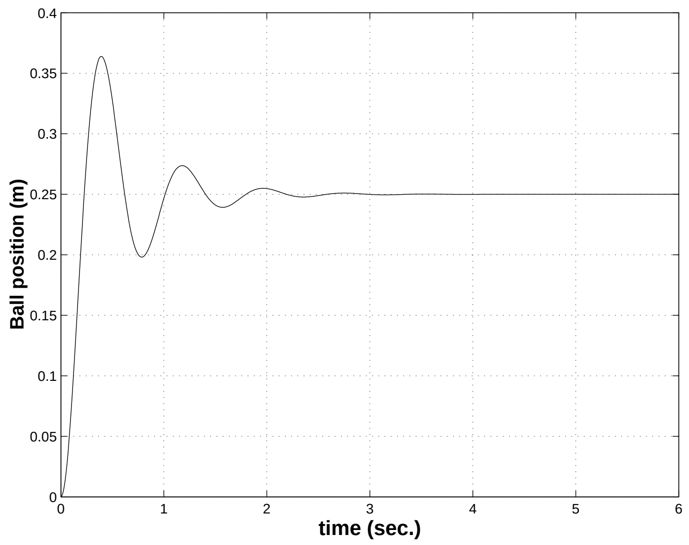

Figure 23: Simulated response of the ball-beam example to a step input.

This small example of classical feedback control does not account for the many advantages of modern feedback control. Despite that, it is adequate to illustrate some features and problems found in the STP. Input variables such as the effort in the STP and the angle θ in the example, can be changed/controlled; state variables required to produce the desired output variables can also be defined for the STP as in the ball-example; and so on. A description of a state model for the STP, is presented in Section 4.4.

A feedback mechanism was required to solve the ball and bean example. Classical feedback theory tells us how to choose the "feedback gains" to change inputs and make the measurable variables attain the desired levels of performance. The same is true for the SDP, i.e., we need to define a mechanism to change the inputs, to correct the disturbances thatb arise during the process, and achieve the desired results. Thus feedback system theory can be applied to achieve this goal, but to do so one needs a good model of the software process.