Sādhanā, Vol. 22, Part 1, February 1997, pp. 121–132. © Printed in India.

A re-entrant line model for software product testing

V V S SARMA[1] and D VIJAY RAO[2]

Department of Computer Science & Automation, Indian Institute of Science Bangalore 560 012, India
[1]Present address: Tata Research Development and Design Centre (TCS), Plot 54B, Hadapsar Industrial Estate, Pune 411 040, India
[2]Present address: CASSA (DRDO), New Thippasandra PO, Bangalore 560 075, India
e-mail: vvs@trishul.trddc.ernet.in; vvs@csa.iisc.ernet.in; vijay@cassa.ernet.in

Abstract. In today's competitive environment for software products, quality is an important characteristic. The development of large-scale software products is a complex and expensive process. Testing plays a very important role in ensuring product quality. Improving the software development process leads to improved product quality. We propose a queueing model based on re-entrant lines to depict the process of software modules undergoing testing/debugging, inspections and code reviews, verification and validation, and quality assurance tests before being accepted for use. Using the re-entrant line model for software testing, bounds on test times are obtained by considering the state transitions for a general class of modules and solving a linear programming model. Scheduling of software modules for tests at each process step yields the constraints for the linear program. The methodology presented is applied to the development of a software system and bounds on test times are obtained. These bounds are used to allocate time for the testing phase of the project and to estimate the release times of software.

Keywords. Software quality; software process modelling; re-entrant lines; software product testing.

1. Introduction

In today's competitive environment for software products, quality has become an increasingly important concern to software development organizations. Quality denotes a multidimensional concept. As an intrinsic product attribute, the quality of software is recognized by the absence of defects. If we view quality from the point of product operation, attributes such as reliability, efficiency, usability and integrity are useful; whereas from the point of view of product transition/revision, parameters such as portability, reusability, inter-operability, and maintainability are important (Ghezzi *et al* 1988).

Several models relating to software quality have been proposed in the literature. These may be broadly classified into three categories, each for a separate purpose (Kan *et al* 1994).

(1) *Reliability models* for reliability assessment and prediction.

(2) *Quality management models* for managing quality during the development process. Quality management models are still in their development and maturing phase. These models emerged from the practical needs of large-scale development projects. The phase-based defect removal model and several tracking models belong to this category.

(3) *Complexity models* and metrics which are used by software engineers for quality assurance purposes. Complexity models explain quality from the internal structure and complexity of the software.

Software reliability modelling is more mature than the other two types. A plethora of software reliability models have been developed over the years but, in spite of the extravagant claims for their efficacy, none can be trusted to give accurate results in all circumstances. An important reason for this is the validity of the assumptions underlying these models.

(1) A detected fault is immediately corrected.

(2) No new faults are introduced during the fault removal process.

(3) Reliability is a function of the number of remaining faults.

(4) Failure rate increases between failures.

(5) Testing is representative of the operational usage.

(6) Software is treated as a blackbox without looking at its structure and the process of its development.

Recently, there has been much emphasis on improving the software development process, with the assumption that this will lead to improved product quality. However, a precursor to improved processes is an understanding of the dynamics of current processes. With respect to software processes, there are two prevailing schools of thought (Bollinger & McGowan 1991):

- International Standards Organization (ISO) 9000 certification, and
- Software Engineering Institute (SEI) assessment based on the capability maturity model (CMM).

Process models for quality ensure the application of process engineering concepts, techniques, and practices to explicitly monitor, control, and improve the software process. However, these models do not yield quantitative measures of parameters such as reliability and usability to denote the quality of the product in the end.

Software development lifecycle is a model of the software process. There are many steps and activities in building a software product. The process followed to build, deliver and evolve the software product from the inception of an idea all the way to delivery and final retirement of the system is called the *software production process* and the order in which these activities are performed defines the lifecycle for the product. Many models which attempt to capture this process, also called the *software lifecycle* models, have been developed. Such models are based on the recognition that software, like any other industrial product, has a lifecycle which extends from its initial conception to its retirement and that its lifecycle must be anticipated and controlled in order to achieve the desired qualities of the product. Dalal *et al* (1993) distinguish between the upstream phases comprising requirements, specifications and design, and downstream phases comprising coding, testing and maintenance of the software development process.

Conventionally, the software process is supposed to proceed sequentially from requirements to specifications, design, code, testing, and then to release. One extreme description of the process of software development conjures up the image of a *waterfall* flowing from requirements successively onto release with no feedback from a succeeding phase to a preceding phase. The other extreme envisions a *spiral* where feedback constantly loops back from a succeeding phase to a preceding phase as repair of the process is needed. In practice, the actual process could lie anywhere in between and one needs to accurately model the flow of software modules. This is analogous to the flow of silicon wafers undergoing processing (such as deposition, photolithography, etching etc.) in a semiconductor manufacturing plant. A study of software faults in the different phases of the lifecycle suggests that a majority of faults occur in the coding phase (Marick 1990) and that coding errors have substantially more severe effects than do design errors. Testing thus occupies a very crucial role in the overall software development process. The purpose of software testing is to detect errors in a program and, in the absence of errors, gain confidence in the correctness of the program. Efforts to improve the effectiveness of testing can yield substantial gains in software quality.

In this paper, we propose a queueing model based on re-entrant lines (figure 1) to depict the process of software modules undergoing testing/debugging, inspections and code reviews, verification and validation, and quality assurance tests before being accepted for use. This is the first model of its kind which depicts the *process* of testing software as seen in the software industry. The model takes into account the structure of the software, the individual modules being distinguished by their criticality in the mission and implementation, their usage in the operational field from profiles and test strategies used for testing these modules. We consider in our model, the notion of imperfect debugging and that new faults can be introduced in the process of imperfect debugging. The paper is organized as follows:

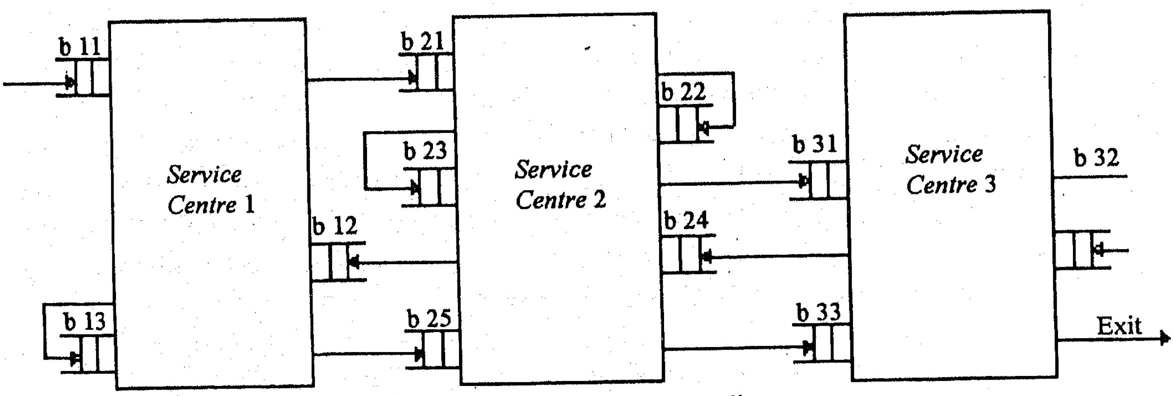

Figure 1. A typical re-entrant line.

Software development lifecycle is described in § 2, re-entrant lines and some important results are discussed in § 3, bounds on test times for software products using re-entrant lines are described in § 4, and a case study to illustrate the methodology is shown in § 5.

2. Software development process modelling

A large software project after requirement analysis and design is given to different programming teams for development. It is assumed that some software engineering methodology is used. The software is divided into modules based on the functions, its size and complexity. These modules after development need to be tested at various stages of the product building. Testing is done by the developers during the coding stage (local/unit testing). The module, after unit testing, is given to an independent test team, not involved in its development, for further testing. This independent test team detects the faults and these modules are sent back to the developers with a log of the tests done and their outcomes. This is also reflected in the configuration control management (CCM) of the project. The developing team then debugs the code and corrects the errors. The same sequence is followed for all the modules of the software. This process continues till the required reliability for the module is achieved or the testing time allotted for it is reached. Different criteria to stop testing have been suggested in the literature (Dalal & Mallows 1989; Musa & Ackerman 1989).

Once these modules are tested, they are integrated and tested for interface errors and inconsistencies across modules. These, along with the libraries and related documentation and standards, form the complete product. The validation of this product is done by an independent verification and validation (IVV) team. Code walkthroughs, inspections and quality assurance tests are done at all stages from coding to acceptance of the software product. These tests defer modules to further testing if they do not conform to requirements/standards prescribed, which would otherwise certify the product for release.

This whole process can be viewed as a multi-class queueing network as depicted in figure 2. The test teams denote the servers and the modules represent the customers who arrive for service (testing). In figure 2, the first team denotes the unit-testing team where the developers locally test the modules during its development, the second server represents the independent test team, the third team denotes integration tests and IVV; and the fourth team, the QA and system testing.

Consider the flow of a tagged module M through such a process. At the first test team, TT1, the module is unit tested by the developers. This module M is tested by an independent test team TT2 and the errors (if any detected and located) corrected by TT1. Unit tested modules arrive for integration and later for verification and validation. Interface errors, non-conformance with requirements, or inconsistent representational formats with some modules causing integration tests to fail result in these modules being sent back to the corresponding teams for correction. Finally, when all the modules are integrated, the system is tested with QA team for checking the process of development and resulting product. The QA team either accepts the product for release or recommends the software to be rectified by the teams. In the following section, we describe a process model to depict the downstream phases of the lifecycle based on re-entrant lines.

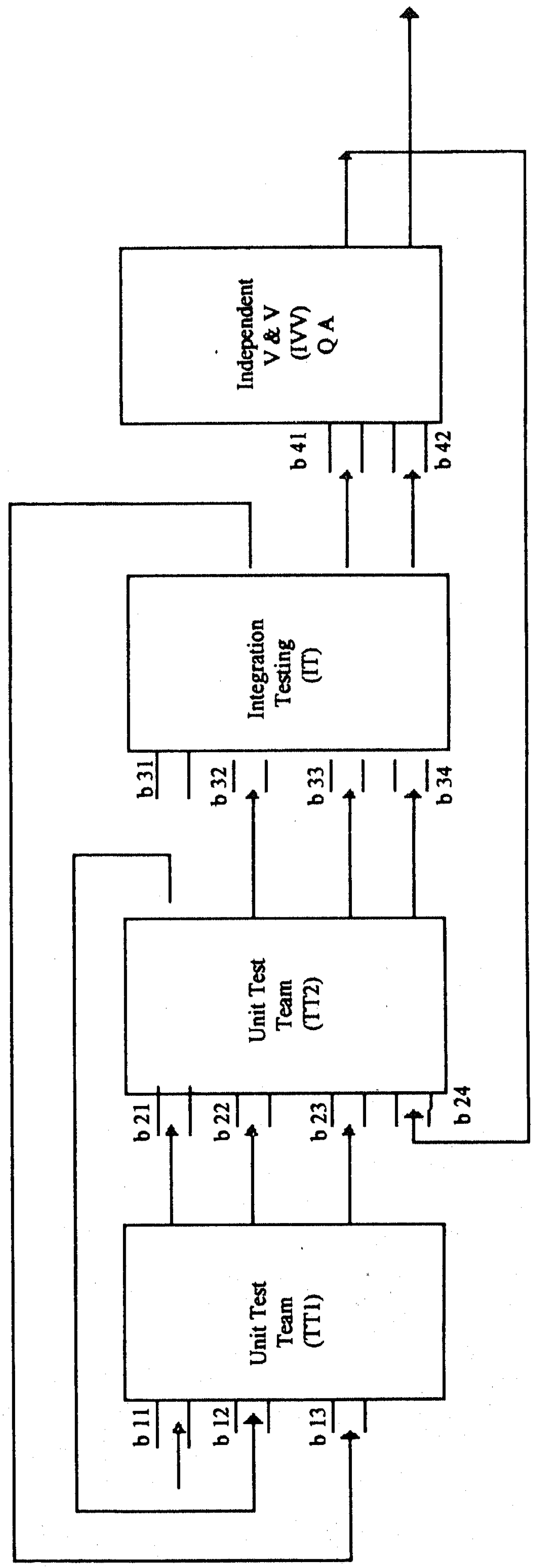

Figure 2. A re-entrant line model for downstream lifecycle phases for a software product with several modules.

3. Model development

Semiconductor wafer manufacturing plants are organized quite differently from traditional assembly lines or job shops. The production process of a silicon wafer consists of imprinting several layers of chemical patterns on the wafer; the final end product obtained is a multi-layered sandwich. Each layer in turn requires several steps of individual processing such as deposition, photolithography, etching, etc. with many of the steps repeated at several of the layers. The machines to perform these individual steps are very expensive. Hence, the machines are not replicated but revisited by the wafers for processing at different layers. The distinguishing characteristic of such a manufacturing system (modelled as a multi-class queueing networks), called a *re-entrant line*, is that the lots revisit several machines at several stages of their life. The main consequence of the re-entrant nature is that several wafers at different stages of their life have to compete with each other for the same machines. Figure 1 shows a re-entrant line with 3 service centres and 11 buffers. Parts enter the system at buffer b_{11} and visit the centres according to a deterministic route as shown. Finished parts emerge from centre 3 after undergoing processing following a wait in b_{33}. Note that each part in this example line visits centre 1 three times, centre 2 five times, and centre 3 thrice. Scheduling in re-entrant lines, input releases and scheduling policies have a significant effect on the performance of this system. Several policies have been studied by Kumar (1994), Lu *et al* (1991) and Khan (1995).

Several researchers have recently come up with analytical methods to obtain upper and lower bounds on the performance of scheduling policies in multi-class Markovian networks (re-entrant lines) (Kumar & Kumar 1994). These methods rely on assuming stability and obtaining a set of linear constraints on the mean values of certain random variables that determine the performance of the system. Augmenting these constraints with others obtained using conservation principles, bounds on performance can be obtained by solving the resulting linear program. Bounds on the mean delay (called cycle time) are obtained with different scheduling policies. The cycle time in software testing process corresponds to the time required to test all the software modules before release to the customer.

In the proposed model for software product testing, servers (machines) denote the test teams and parts (silicon wafers) denote the software modules undergoing testing and correction. Due to the large number of modules at different stages of testing, test teams also need to schedule their tasks to select the next module to test.

Consider a set of $\{1, 2, 3, \ldots, S\}$ of S test teams consisting of professionals and developers of the code. Modules are classified based on their criticality and usage (from profiles). Modules of similar reliability requirements enter the system for testing at a test centre $s(1) \in \{1, 2, 3, \ldots, S\}$ where they are labelled as of class type C_i. Let C_L class of modules being tested at $s(L)$ be the last set of tests done on these modules. The sequence $\{s(1), s(2), \ldots, s(L)\}$ is the route followed by the modules for tests. These modules visit the next team $s(2)$ after being tested at $s(1)$ and so on. We shall allow for the possibility that $s(i) = s(j)$ for some classes $i \neq j$ and accordingly call this type of system a re-entrant line (Kumar 1993).

For this system we assume:

(1) Modules arrive into the system for testing according to a Poisson process with rate λ;

(2) The mean time to test for every class C_i is $1/\mu_i$ and the times to test are distributed exponentially.

The first team denotes unit/local testing which is done by the programmer himself/herself. These modules take $1/\mu_1$ amount of time for the local testing. After this testing, the module is passed on to Team 2 which is a peer test team, not involved in the development of the module. Any bugs located by this team are recorded in the error log and sent back to the developers for correction and testing thereof. The time for testing this module would now be governed by the mean time to test for modules of class C_2. In this fashion, when the modules are approved by the Teams 1 and 2, it passes on to Team 3 denoting Integration testing and System testing. Team 4 denotes Product testing and QA which checks for the process of software development and the product developed. This team either accepts the product in which case it is delivered to the user along with the proper documentation, or it sends back particular parts of the product which have non-conformance reports to the design team or for further testing. This feedback defines the re-entrant path for the module. This completes one cycle of the downstream process for the software. Due to non-conformance of some modules to the specifications/standards, the product release date is shifted till another cycle of the process is completed. However, in Cycle 2, the mean time to test for some classes is less than that of Cycle 1, due to the learning factor, experience gained and familiarity with the system to generate efficient test cases which maximize the coverage. This is analogous to the product-in-a-process approach suggested by Laprie (1993) to develop families of software. The path followed by modules demanding different levels of quality in this process is varied. Based on this model, we compute bounds on mean test time for modules.

4. Bounds on test times: The LP approach

Consider a strategy to select the next module for testing, which is –

(1) *Nonidling*: If there is any module to be tested then the test team does not stay idle;

(2) *Stationary*: The decisions to select the next module depends only on the number of modules of different classes in the system (Lu *et al* 1991; Kumar 1993, 1994).

Let us rescale time so that $\lambda + \sum_{i=1}^{L} \mu_i = 1$. We use uniformisation in which we sample a continuous time system to obtain a discrete time system with the same steady-state behaviour. We sample the system at all service completion times, as well as at the arrival times of new modules to the system for testing. Let $\{\tau_n\}$ be the sequence of such random sampling times and let F_{τ_n} denote the σ-field generated by the events up to time τ_n. Let $X_i(t)$ denote the number of modules of class C_i at time t. Also, let $W_i(\tau_n) = 1$ if the testing team at $\sigma(i)$ is working on the module of class C_i at time t, and 0 otherwise. We take all processes to be right continuous, and thus $X_i(\tau)$ is the state after the nth event, while due to the stationarity of the strategy chosen, $W_i(\tau_n) = 1$ implies that the team $\sigma(i)$ is busy working on C_i class of modules in the interval $[\tau_n, \tau_{n+1})$. Let us denote

$$X^T(\tau_n) = (X_1(\tau_n),\ X_2(\tau_n), \ldots, X_L(\tau_n)). \tag{1}$$

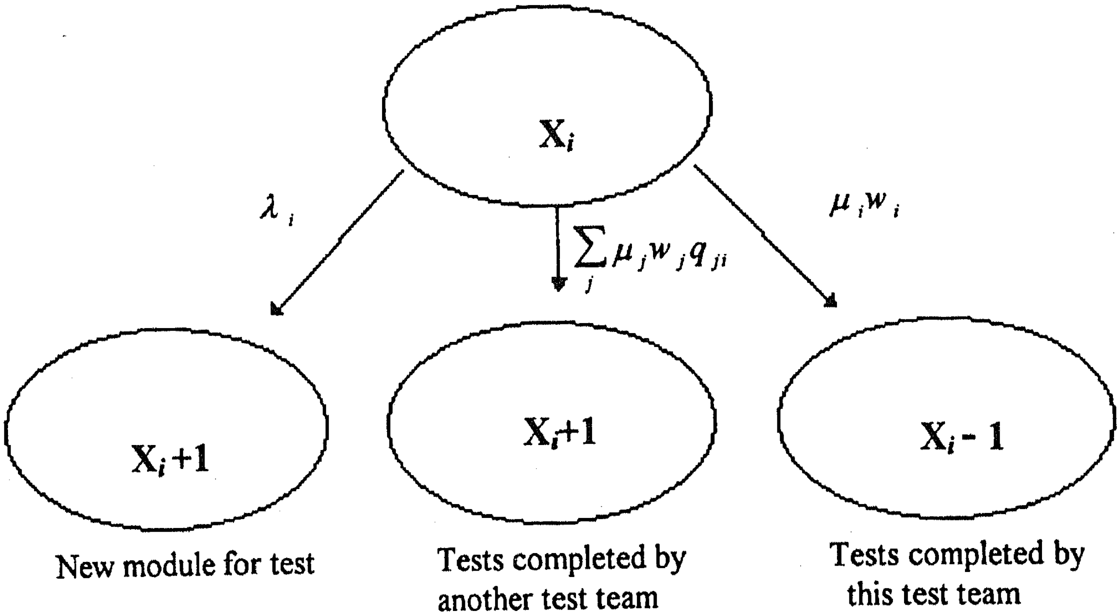

Figure 3. State transitions for a class C_i of modules.

In the steady state,

$$E[X^T(\tau_{n+1}).Q.X(\tau_{n+1})] = E[X^T(\tau_n).Q.X(\tau_n)], \tag{2}$$

for every symmetric matrix Q. We presume that the steady-state distribution has a finite second moment on the total number of modules at each buffer. For this equation to hold, we need

$$E[X_i(\tau_{n+1}).X_j(\tau_{n+1})] = E[X_i(\tau_n).X_j(\tau_n)] \quad \text{for } 1 \le i,\ j \le L. \tag{3}$$

Now consider the implication of the equality

$$E[X_1^2(\tau_{n+1})] = E[X_1^2(\tau_n)].$$

From the state transitions for the buffer i shown in figure 3, we have

$$\begin{aligned} X_i(\tau_{n+1}) &= X_i(\tau_n) + 1\text{: exogeneous arrival to } C_i \text{ at } \tau_{n+1},\\ &= X_i(\tau_n) + 1\text{: previous class tests completed},\\ &= X_i(\tau_n) - 1\text{: current class tests completed},\\ &= X_i(\tau_n) \quad\text{: otherwise}. \end{aligned}$$

Suppose every class C_i has an exogenous arrival process, which is Poisson with rate λ_i. Also suppose that with probability q_{ij}, a module passes from class C_i to C_j.

From the equality equation, $E[X_1^2(\tau_{n+1})] = E[X_1^2(\tau_n)]$, and using the stationarity policy of the strategy used, we get the following equality constraints:

$$2\lambda_i\left(\sum_{j\in I(i)} z_{ji}\right) + 2\sum_{j=1}^{L}\mu_j q_{ji} z_{ji} - 2\mu_i z_{ii} + 2\mu_i \rho_i = 0, \tag{4}$$

where $z_{ij} = E[W_i(\tau_n).X_j(\tau_n)]$

$$\lambda_i\left(\sum_{k\in I(j)} z_{kj}\right) + \lambda_j\left(\sum_{k\in I(i)} z_{ki}\right) + \sum_{k\neq j}\mu_k q_{ki} z_{kj} + \sum_{k\neq i}\mu_k q_{kj} z_{ki}$$
$$+ \mu_j q_{ji}(z_{jj} - z_{ji} - \rho_j) + \mu_i q_{ij}(z_{ii} - z_{ij} - \rho_i)$$
$$- \mu_i(1 - q_{ij})z_{ij} - \mu_j(1 - q_{ji})z_{ji} = 0.$$

Now using the nonidling policy, we get the following inequality constraints:

$$\sum_{\{j|\sigma(j)=\sigma\}} z_{ji} \leq \sum_{j\in I(i)} z_{ji}, \quad \text{for } i = 1, \ldots, L; \quad \sigma = 1, \ldots, S \text{ with } \sigma \neq \sigma(i), \tag{5}$$

and the nonnegativity constraints

$$z_{ij} \geq 0 \quad \text{for } i, j = 1, \ldots, L. \tag{6}$$

If the scheduling strategy is stationary and nonidling with a steady-state distribution possessing a finite second moment, then the mean number of modules in the system at various stages of testing is bounded above by

$$\max \sum_i \sum_{j\in\sigma(i)} z_{ji}, \tag{7}$$

and below by

$$\min \sum_i \sum_{j\in\sigma(i)} z_{ji}. \tag{8}$$

Equations (7) and (8) denote the bounds on the number of modules in the system. Using Little's law, $L = \lambda W$ (Little 1961) and assuming that the arrival rate of modules to test is constant, we obtain the bounds on testing time for the modules.

5. Examples

Example 1. In this section, we consider the development of a re-entrant line based software process model for a firm executing a software project of moderate size (needing a few person months of effort). It is identified at the preliminary design level that the software is made up of 40 modules of similar complexity. The underlying re-entrant line model is shown in figure 4. It is assumed that there are two programming and testing teams. Software modules are first unit tested by the developers (Team 1) and Team 2 acts as an independent test team for these modules.

For simplicity, we assume that new modules arrive for testing by Team 1 with rate λ and the route followed by all modules in the re-entrant line model is deterministic. A class C_i of modules takes $(1/\mu_i)$ person hours to test a module. The linear program to bound the mean number of modules in the re-entrant line (figure 4) is:

$$\min[z_{11} + z_{31} + z_{22} + z_{42} + z_{13} + z_{33} + z_{24} + z_{44}]$$

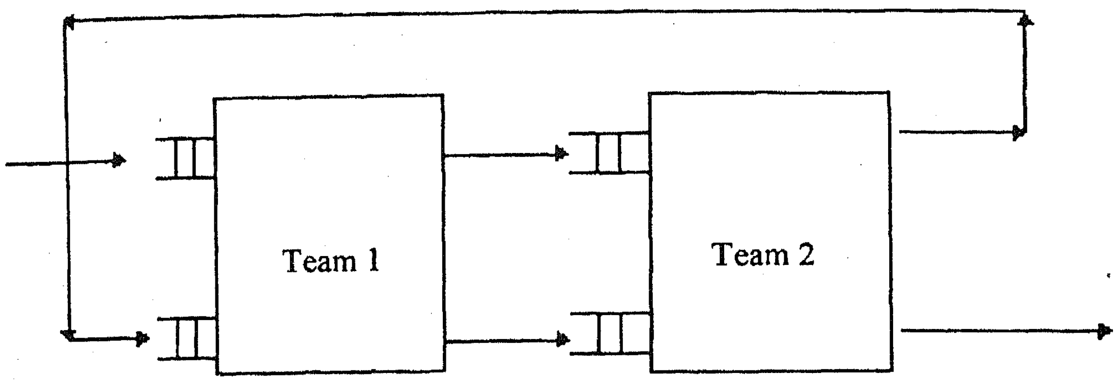

Figure 4. A re-entrant line model for a software testing process.

and

$$\max[z_{11} + z_{31} + z_{22} + z_{42} + z_{13} + z_{33} + z_{24} + z_{44}].$$

The equality, inequality and non-negativity constraints are given by (4), (5) and (6) respectively. Defining $\rho_i = \lambda/\mu_i = 0.25$ and solving the linear program, we obtain bounds on test times as [4.0–4.5] person months. With these estimates of test times, we can allocate approximately 18 person weeks for the testing phase of this project.

Example 2. If the 40 modules in the software system of example 1 are classified based on their criticality and usage, with more test time allocated to the high usage and critical modules, we can obtain realistic values for bounds on test times. We classify the modules based on the criticality of their function in the mission and the usage (from profiles) in the use environment. The *Criticality-Usage* matrix is formed for the modules of the system. This was used to decide the *class* to which the module enters and hence the testing time. These data are summarized in table 1.

In the above matrix, modules of $\{CU(1,1)\}$ are made members of class C_1, modules of $\{CU(1,2),\ CU(2,1),\ CU(2,2)\}$ are made members of class C_2 and modules in $\{CU(1,3),\ CU(2,3),\ C(3,1),\ CU(3,2),\ CU(3,3)\}$ are made members of C_3, as these are critical and frequently used modules. The testing times are varied accordingly in the ratio of 1:2:4 for the modules of classes C_1: C_2: C_3. The linear program is solved to obtain the bounds on the test times for the modules of different criticality and usage. The results obtained are summarized in table 2.

Table 1. The criticality-usage matrix $CU(i, j)$.

	Criticality		
Usage	Low	Medium	High
Low	6	10	2
Medium	4	6	2
High	4	3	3

Table 2. Bounds on test times for example 2 (§ 5).

Class	Mean test time in person weeks (bounds)
C_1	[3, 4]
C_2	[8, 11]
C_3	[19, 23]

Total test time: [30, 38] person weeks or [7.5–9.5] person months (Assume 4 person weeks in a person month)

With these estimates of test times, we can allocate approximately 9.5 person months of testing. If the milestone for completion of coding is set at the end of 14th month, then we can allocate the testing and verification phase to end by the 24th month from the start of the project.

6. Conclusions and discussion

A process model which depicts the downstream phases of the software life cycle modelled as a re-entrant line is presented. Further, based on this model, a method to compute bounds on test times of software is presented. Due to priority test scheduling of modules, the re-entrant model is not of product form and hence not amenable to closed form solutions for steady-state analysis. Bounds on test times are obtained by considering the state transitions for a general class of modules which leads to a linear programming model. Scheduling of software modules for test at each process step yields the constraints for the linear program. From the bounds on the test times, the product release times are obtained. We illustrate the methodology using an application for which bounds on test times are obtained. For modules of varying criticality-usage factor, we observe that the *test times are not scaleable*.

In software development applications, a module's route through test teams is not the same for all modules and is not deterministic. The current model can be extended to reflect this situation with the introduction of path profiles and a route matrix for the modules (Vijay Rao 1995). This model can also be used to decide on the release times of software with a specified reliability measure (Vijay Rao 1995).

The authors wish to thank the anonymous referees for their useful comments which helped in improving the examples of § 5, and Dr N K Srinivasan and Prof Y Narahari for useful discussions.

References

Bollinger T B, McGowan C 1991 A critical look at software capability evaluations. *IEEE Software* 7: 25–41

Dalal S R, Mallows C L 1989 When should one stop testing? *J. Am. Stat. Assoc.* 83: 872–875

Dalal S R, Horgan J R, Kettenring J R 1993 Reliable software and communication: Software quality, reliability and safety. *Proc. 15th Int. Conf. Software Engineering* (Los Alamitos, CA: IEEE Comput. Soc. Press) pp 425–435

Ghezzi C, Morzenti A, Pezze M 1988 On the role of software reliability in software engineering. *Software reliability modelling and identification* (ed.) S Bittanti (Berlin: Springer-Verlag) pp 1–41

Kan S H, Basili V R, Shapiro L N 1994 Software quality: An overview from the perspective of total quality management. *IBM Syst. J.* 33: 4–18

Khan L M 1995 *Performance analysis of scheduling policies in stochastic re-entrant lines.* Ph D dissertation, Indian Institute of Science, Bangalore

Kumar P R 1993 Re-entrant lines. *Queueing Syst. Theor. Appl.* 13: 87–110

Kumar P R 1994 Scheduling queueing networks: stability performance analysis and design. *Proc. IMA workshop on stochastic networks* (Berlin: Springer-Verlag)

Kumar S, Kumar P R 1994 Performance bounds for scheduling queueing networks. *IEEE Trans. Autom. Control* 39: 1600–1611

Laprie J C 1993 For a product-in-a-process approach to software reliability evaluation. PDCS Tech. Report ESPRIT-BRA-6362-PDCS2, University of Newcastle upon Tyne

Little J D C 1961 A proof of the queueing formula $L = \lambda W$. *Oper. Res.* 9: 383–387

Lu S C H, Ramaswamy D, Kumar P R 1991 Efficient scheduling policies to reduce mean and variance of cycle-times in semiconductor manufacturing plants. *IEEE Trans. Semiconductor Manuf.* 7: 374–388

Marick B 1990 A survey of software faults. Report No. UIUCDCS-R-90-1651, Dept. of Computer Science, Univ. of Illinois, Urbana, Champaign

Musa J D, Ackerman A F 1989 Quantifying software validation: When to stop testing? *IEEE Software* 5: 19–27

Vijay Rao D 1995 *Estimation of software release times based on a queueing model for software testing*. MSc (Eng.) thesis, Indian Institute of Science, Bangalore