Requirements Volatility and Defect Density

Yashwant K. Malaiya & Jason Denton

Colorado State University
Department of Computer Science
Ft. Collins, CO 80523
E-mail: malaiya|denton@cs.colostate.edu

Abstract

In an ideal situation the requirements for a software system should be completely and unambiguously determined before design, coding and testing take place. In actual practice, often there are changes in the requirements, causing some of the software components to be redesigned, deleted or added. Higher requirement volatility will cause the resulting software to have a higher defect density.

In this paper we analytically examine the influence of requirement changes taking place during different times by examining the consequences of software additions, removal and modifications. We take into account interface defects which arise due to errors at the interfaces among software sections. We compare the resulting defect density in the presence of requirement volatility, with defect density with that would have resulted in an ideal situation where initial requirements are perfect. The results show that if the requirement changes take place closer to the release date, there is a greater impact on defect density. In each case we compute the defect equivalence factor representing the overall impact of requirement volatility. Further work required to obtain an overall model that can be used in an empirical model for defect density, is mentioned.

1 Introduction

Defect density is an important measure of software quality, one which is often used as an acceptance criteria for a piece of software. For this reason it is desirable to understand how various aspects of the development process impact defect density, so they can be controlled or at least used to gain a better understanding of product reliability. The maturity of the development process, the skill of the programmers involved, and the complexity of the program all play a significant part in the defect density of a program [3]. Studies suggest that changes to the requirements specification also have a significant impact on defect density [13].

Requirements volatility is a measure of how much program's requirements change once coding beings. Projects for which the requirements change greatly after coding begins have a high volatility, while projects whose requirements are relatively stable have a low volatility [2, 10,

11, 12]. The analysis presented here shows that the time at which requirements changes are made is a significant factor in program defect density. Changes made late in the development cycle can not only waste development resources, but also reduce the overall testing effectiveness.

Requirements specifications are often written in natural language. Even when more precise techniques are used these specifications tend to change as program development and testing progresses. Often new requirements are added and existing requirements are modified or deleted. As a response to these changes, the program is also modified. Here we present an analysis of how requirements volatility affects a project by examining program evolution in terms of changes to the code base. This is a topic first examined by Musa et al. in [7], where they examined how continuing program evolution violates the usual assumptions made by the standard software reliability growth models, and how corrections to the procedures can be made. Changes made to requirements must eventually be reflected in the code, and developing organizations can, over time, develop a feel for how particular changes in their requirements specifications impact their code. In this paper we related how changes to the code effect the over all defect density. In our analysis we assume that that software has been modified as a response to the changing requirements, however the modification process is imperfect. To keep analysis tractable, we assume that debugging for individual defects is perfect. In actual practice, a fraction of the bugs are incorrectly debugged. Ohba and Chou have shown that in such a case a reliability growth model is still applicable, although imperfect debugging cause some variation in the parameter values [8].

In this paper we evaluate the impact of code changes on the defect density by considering four separate cases. In the next section, we consider the the simplest case when a block of code is replaced by a newly developed block of the same size. We mention two significant assumptions made and show how they can be relaxed for more accurate calculation. In section 3, we consider those cases where a section of the software is added, when a new component is added and when a component is modified. In each case we compute the resulting additional defect density. We also obtain a multiplicative factor to obtain *equivalent initial defect density*. Finally we suggest a preliminary model that can be used as part of a static model for estimating defect density.

2 Same Sized Code Replacement

Here we consider the relatively simple problem of estimating defect density in a software system, when a component is replaced by a new block of the same size. We assume that the original system had a defect density of D_0 at the time $t_0 = 0$. Here the instant t_0 can be regarded as the time when the system enters a specific testing phase. At time t_1 a component is replaced by new code. We assume that the new code enters with defect density D_0.

The exponential reliability growth model [5] assumes that the rate of defect removal $\frac{dN}{dt}$ is proportional to the number of defects $N(t)$ present at time t.

$$\frac{dN(t)}{dt} = \beta_1 N(t) \tag{1}$$

where t is testing time. Note that t may or may not be closely related to calendar time depending on how resources are allocated in a project.

It can be shown that the parameter β_1 can be expressed as [4]

$$\beta_1 = \frac{kr}{SQ} \tag{2}$$

where S is the total number of source instructions, Q is the number of object instructions per source statement and r is the instruction rate of the CPU used. The parameter k is called fault exposure ratio, which has been found to be in the range of 1×10^7 to 10×10^7.

If N_0 is the number of defects present at time t_0, then from equation 1

$$N(t) = N_0 e^{-\beta_1 t} \quad t_0 \leq t \leq t_1 \tag{3}$$

or equivalently in terms of defect density $D(t)$

$$D(t) = D_0 e^{-\beta_1 t} \tag{4}$$

since $D(t)S = N(t)$.

Now let us assume that the fraction of the code replaced is p. Thus at t_1, the number of defects remaining in the old code is

$$N_1 = N_0(1-p)e^{-\beta_1 t_1} = D_0 S(1-p)e^{-\beta_1 t_1} \tag{5}$$

and the number of defects in the new code is

$$N_2 = D_0 Sp \tag{6}$$

If there is no future evolution of the software, except for removal of defects found, then then

$$N(t) = (N_1 + N_2)e^{-\beta_1 (t-t_1)} \qquad t_1 \leq t \tag{7}$$

Notice that β_1 depends on the size, however since that total size has remained the same, β_1 remains unchanged. Equation 7 can be written as

$$N(t) = D_0 S[(1-p)e^{-\beta_1 t_1} + p]e^{-\beta_1 (t-t_1)} \qquad t_1 \leq t \tag{8}$$

If testing is to be terminated at time t_f, the final number of remaining defects is

$$N_f = D_0 S[(1-p)e^{-\beta_1 t_1} + p]e^{-\beta_1 (t_f - t_1)} \tag{9}$$

and the final defect density is

$$D_f = D_0[(1-p)e^{-\beta_1 t_1} + p]e^{-\beta_1 (t_f - t_1)} \tag{10}$$

The influence of the change occurring at time t_1 is shown in Figure 1. While further testing reduces the number of new defects injected at t_1, the software still ends up with a higher number of defects at the end of testing at time t_f.

We can compare D_f with D_{fi}, final defect density in the ideal case. The difference $(D_f - D_{fi})$ gives the additional defect density D_{add} due to requirement volatility.

$$D_{add}(t_1) = D_0 p[e^{-\beta_1 (t_f - t_1)} - e^{-\beta_1 t_f}] \tag{11}$$

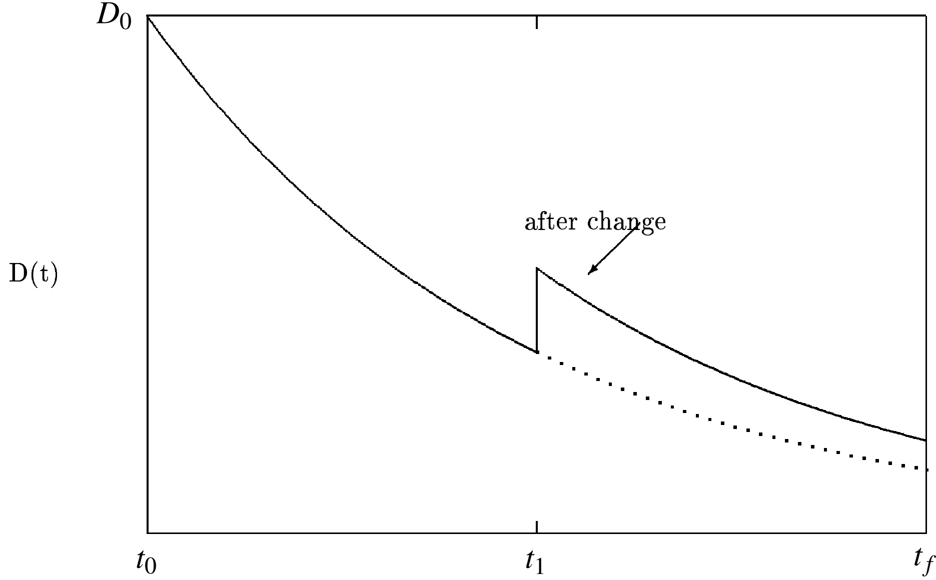

Figure 1. The impact of change on Defect Density

For a given stopping time t_f, D_{add} is a function of $t_f - t_1$, ie. closeness of the change time to the stopping time. This is illustrated in Figure 2. The values used for the plots in Figure 2 assume an initial defect density of 20/KLOC and it is assumed that testing for 3000 time units will reduce the defect density to approximately one tenth.

Equation 10 shows that the change at t_1 results in higher defect density D_f. Here let us define *equivalent initial defect density* D_0' which would have yielded the same D_f without the change at t_1. We can rewrite equation 10 as

$$D_f = D_0[(1-p)e^{-\beta_1 t_1} + p]e^{\beta_1 t_1}e^{-\beta_1 t_f} \tag{12}$$

Hence

$$D_0' = D_0[(1-p)e^{-\beta_1 t_1} + p]e^{\beta_1 t_1} \tag{13}$$

Thus the change is effectively equivalent to multiplying the defect density by the *defect equivalence factor (DEF)* e_d

$$e_d(t_1) = (1-p) + pe^{\beta_1 t_1} \tag{14}$$

Figure 3 gives a plot of e_d againt t_1, the point in time when the change takes place. Figure 4 gives a table of e_d for different values of t_1 and p, the fraction of the code involved. The table shows that replacing 10% of the code at time 500 causes only a 4.2% change in the defect density whereas the same change at time 2500 causes a 47.5% change. The influence of p is linear, a p=5% change at t_1=1500 results in a 9.3% increase, a p=15% change causes 27.9% increase.

In the above discussion we have made two simplifying assumptions. Here we will see that it is possible to obtain more accurate expressions.

Assumption 1: We assumed that the testing time duration, from 0 to t_f is fixed. In actual practice, the exact effort needed for developing the new code (and for separately testing it to get its defect density to D_0) may subtract from the available time. We can assume that this additional

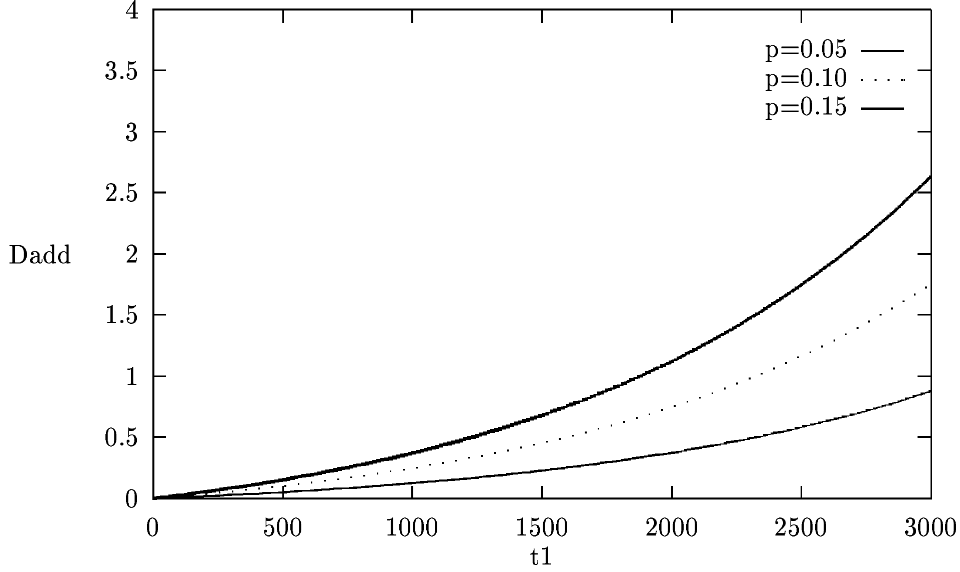

Figure 2. Additional defect density due to change at t_1

effort is proportional to the size of the code replaced. Then the duration t_f is effectively replaced by t'_f such that

$$t'_f = t_f - a \times p \times s \tag{15}$$

where the parameters a would depend on the specific software development and testing process.

Assumption 2: We assume that the number of defects associated with the new code is proportional to its size. This means that we are assuming that the number of defects related to the coupling of the old code to the new code is negligible. The degree of coupling is perhaps measured by the number of variables passed. If a module with a well defined interface with the rest of the code is replaced with new code, then the number of new interface defects introduced will be small. However, if the new code interacts with old code using a large number of variables, then the number of interface defects will be significant.

Let us assume that the number of variables passed from the old code to the new code and vice versa is m. Then the number of interface defects N_{int} can be given by

$$N_{int} = b_1 m \tag{16}$$

where b_1 is a constant of proportionality. Then we can rewrite equation 8 as

$$N(t) = D_0 S[(1-p)e^{-\beta_1 t_1 + p}]e^{-\beta_1 (t - t_1)} + b_1 m \tag{17}$$

The correction term will be significant if the number of variables is very large, or if the rest of the software has a very low defect density.

3 Code addition, removal and modification

In the previous case, the overall software size had remained unchanged. Here we consider the cases when the software size changes as a response to the requirement volatility. We the parameter β_1 of the exponential model, depends on the size. In case of code modification, the number of interface defects can become quite significant.

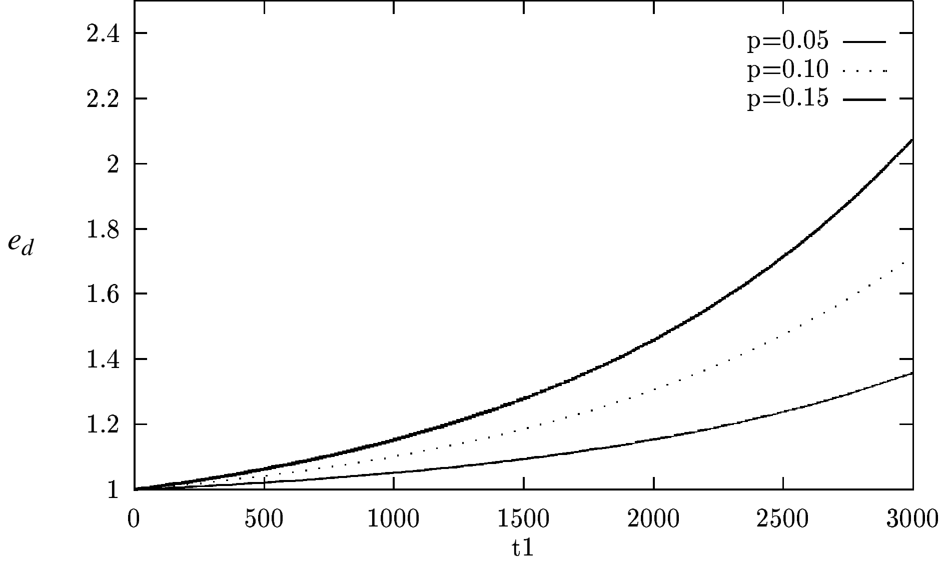

Figure 3. DEF (Defect equivalency Factor) due to code replacement

	Replacement time t_1						
Fraction p	0	500	1000	1500	2000	2500	3000
0.05	1.000	1.021	1.051	1.093	1.153	1.238	1.358
0.10	1.000	1.042	1.101	1.186	1.306	1.475	1.717
0.15	1.000	1.063	1.152	1.279	1.458	1.713	2.075

Figure 4. DEF e_d due to code replacement

3.1 Addition of New Code

Let us now consider the case when at time t_1 new code is added, perhaps to implement additional functionality. Let us assume that the size of the added code is p times the size of the original code, and when the addition occurs the defect density of the added code is D_0. Then at time t_1, the total number of defects in the system is

$$N(t_1) = S_0 D_0 e^{-\beta_1 t_1} + S_0 p D_0 \tag{18}$$

Here we should note that the parameter β_1 depends on program size, as given by equation 2. The corresponding parameter β_{1a} for the altered system is given by

$$\beta_{1a} = \frac{kr}{(1+p)SQ} = \frac{1}{1+p}\beta_1 \tag{19}$$

Then we can write

$$N(t) = D_0 S[e^{-\beta_1 t_1} + p]e^{-\beta_{1a}(t-t_1)} \qquad t_1 < t \tag{20}$$

In an ideal case the added functionality should have been present from the beginning, and the number of defects $N_I(t)$ would have been

$$N_I(t) = D_0 S(1+p)e^{-\beta_{1a} t} \tag{21}$$

Thus the added defect density due to the requirement volatility is

$$D_{add}(t_1) = \frac{1}{s(1+p)}[N(t_f) - N_1(t_f)] \tag{22}$$

$$= \frac{D_0}{1+p}[e^{-(\beta_1-\beta_{1a})t_1} + pe^{\beta_{1a}t_1} - (1+p)]e^{-\beta_{1a}t_f} \tag{23}$$

Figure 5 shows the additional defect density due to adding code later at time t_1 instead of including it from the beginning. This suggests that if code is added immediately after time $t = t_0$, the influence on resulting defect density is relatively small. The defect equivalency factor is this case is

$$e_d = \frac{(e^{-\beta_1 t_1} + p)e^{\beta_{1a}t_1}}{1+p} \tag{24}$$

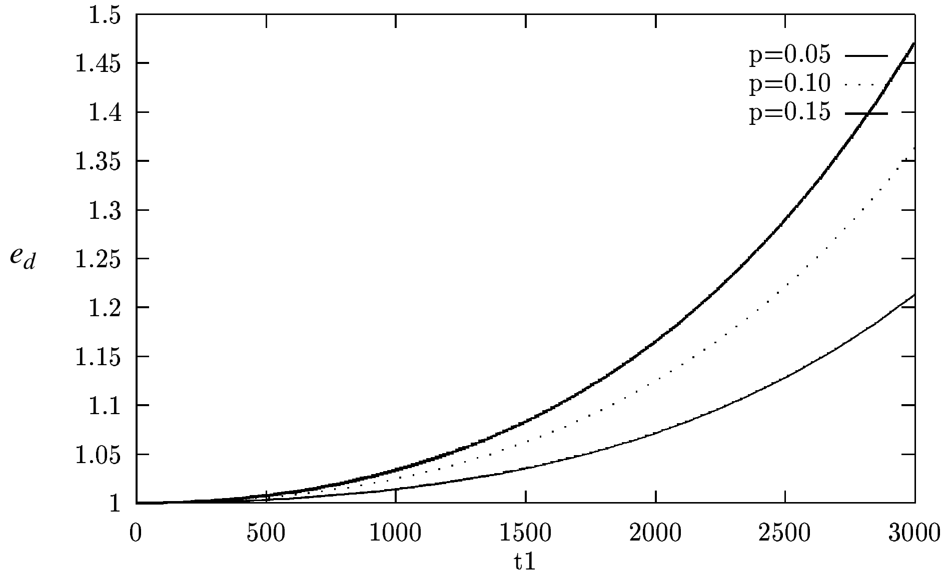

Figure 5. DEF due to adding code at t_1

Here no correction would be required to t_f because we are assuming that the additional code is required and it is added at time t_1 rather than at $t = 0$; and thus no additional development time is needed. We have assumed here that the number of interface defects is small compared to internal defects of the software being added. This may not be valid in cases where the added modules may have significantly low defect density. This may happen when the new code being added is drawn from library modules or represents reused code. In that case, interface defects will be significant and may even dominate the additional internal defects. We can then rewrite 20 as

$$N(t) = [D_0 S e^{-\beta_1 t_1} + D_0' Sp + N_{int}]e^{-\beta_{1a}(t-t_1)} \qquad t_1 < t \tag{25}$$

and the expression for e_d would be

$$e_d = \frac{(D_0 S e^{-\beta_1 t_1} + D_0' Sp + N_{int})e^{\beta_{1a}t_1}}{D_0 S(1+p)} \tag{26}$$

where D'_0 is the defect density of the added code. N_{int} may be estimated using the approach given in the next section.

3.2 Code Removal

The third case we consider is when part of the code is removed, perhaps due to deletion of some requirements. In the ideal case, this code would not have been added in the first place. Removing a section of the code will eliminate all defects in it. However, all linkage between existing code and deleted code must be removed or redirected. Mistakes in this part of the removal process will generate N_{int} additional defects. We can expect that

$$N_{int} = b_2 m \tag{27}$$

where m is the number of linkage variables affected and b_2 is a parameter. For a preliminary computation, we can assume that m is proportional to the size of code removed (pS) and the parameter b is proportional to D_0. We can thus assume that

$$N_{int} = c_1 PSD_0 \tag{28}$$

where the parameter c_1 depends to relative occurrence rate of interface defects as opposed to internal defects. We can expect c_1 to be significantly less than one.

If at time t_1, fraction p of the entire code is removed, then the number of remaining defects will be

$$N(t_1) = D_0 S(1-p)e^{-\beta_1 t_1} + N_{int} \tag{29}$$

Since the code size is now only $S(1-p)$, the applicable parameter β_{1d} is given by

$$\beta_{1d} = \frac{kr}{S(1-p)Q} = \frac{\beta_1}{1-p} \tag{30}$$

and the defect density will be given by

$$D(t) = \left[D_0 e^{-\beta_1 t_1} + \frac{N_{int}}{S(1-p)}\right] e^{-\beta_{1d}(t-t_1)} \tag{31}$$

$$t > t_1 \tag{32}$$

In an ideal case, the deleted code would not have been present from the beginning, and the defect density would have been

$$D_1(t) = D_0 e^{-\beta_{1d} t} \tag{33}$$

Then, the additional defect density is

$$D_{add}(t_1) = D_0(e^{-\beta_1 t_1 + \beta_{1d} t_1} - 1)e^{-\beta_{1d} t_f} + \frac{N_{int}}{S(1-p)} e^{-\beta_{1d}(t_f - t_1)} \tag{34}$$

The density equivalency factor can be obtained as

$$e_d = \left[e^{-\beta_1 t_1} + \frac{N_{int}}{S(1-p)} \frac{1}{D_0}\right] e^{\beta_1 t_1} \tag{35}$$

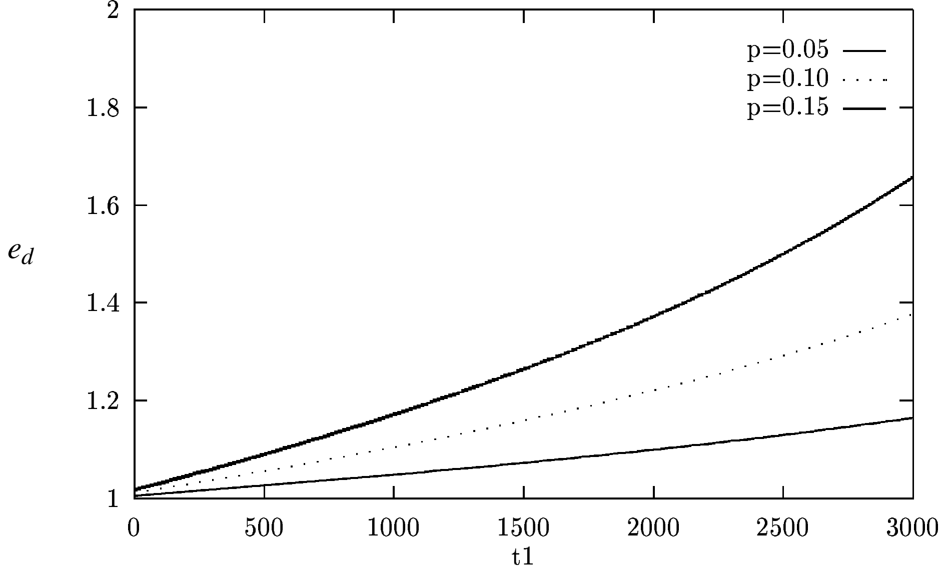

Figure 6. DEF due to code removal vs. t_1

Figure 6 shows an exponential relationship between the time and the defect equivalency factor. In Figure 7 three plots show the effect of variation of p with t_1 equal to 1000, 2000 and 3000 time units respectively. The plots show that at the beginning p does not have much influence. However when removal is done closer to t_f, the fraction has a significant impact on defect density. Figure 8 shows variation in DEF due to change in parameter c_1. For changes made very early, DEF remains relatively unaffected by changes in c_1, however closer to t_f, there is linear dependence.

Here these equations do not take into account the fact that not implemented the unneed code would have saved the effort which would allow a larger testing time t_f.

3.3 Modified Code Block

Here we consider the relatively complex case when a part of the code is modified as a response to requirement changes. The modification will in general include removing some instructions, adding some instructions, and modifying some instructions. Let us assume that the removed and added instructions represent fractions p_1 and p_2 of the original code size S respectively. Let us assume that modifying instructions amounts to replacing them with new instructions. The errors introduced at time t_1 are contributed due to new instructions added as well as due to improper handling of linkage. Let us assume that the number of linkage instances affected is m. Because a variable can be redefined and used many times, m can significantly exceed the number of variables involved.

The number of defects at time t_1 is given by

$$N(t_1) = S(1-p_1)D_0e^{-\beta_1 t} + p_2 S D_0' + N_{int} \tag{36}$$

where N_{int} is the number of interface defects given by

$$N_{int} = b_3 m \tag{37}$$

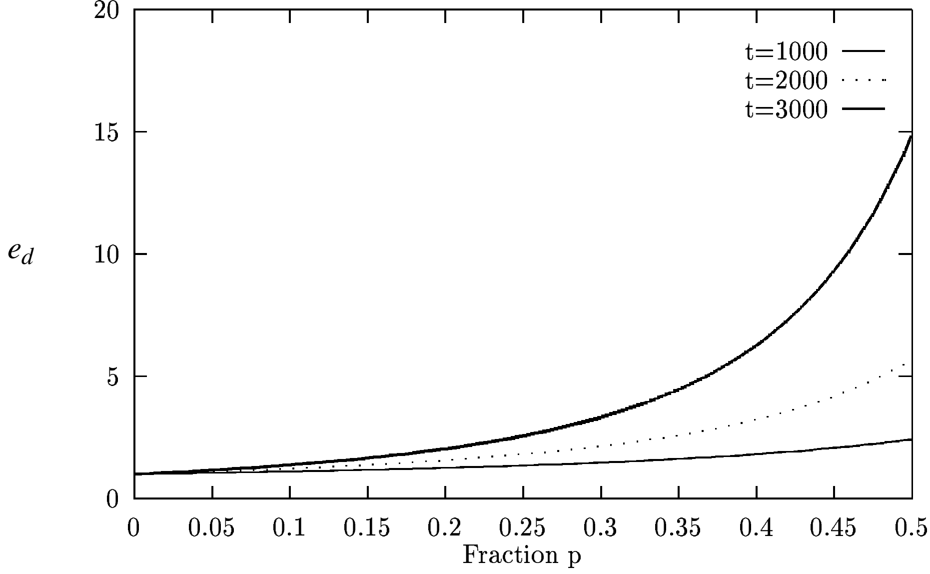

Figure 7. DEF due to code removal vs. p

Again for preliminary calculations, we can assume that m is proportional to the size of software added plus the size of software removed $(p_1 + p_2)S$ and parameter b_3 is proportional to D_0. We can then write

$$N_{int} = c_2(p_1 + p_2)SD_0 \tag{38}$$

where the value of parameter c_2 is likely to be higher than c_1 for the previous case because of a higher degree of linkage.

Using Equation 36, we can write,

$$N(t) = [S(1-p_1)D_0e^{-\beta_1 t} + p_2SD_0' + N_{int}]e^{-\beta_{1m}t-t_1} \qquad t > t_1 \tag{39}$$

where D_0' is the defect density of the new code inserted. We assume that it is inserted without any prior testing and hence would have a defect density higher than D_0. Also,

$$\beta_{1m} = \frac{Kr}{S(1-p_1+p_2)Q} = \frac{\beta_1}{1-p_1+p_2} \tag{40}$$

In the ideal case, all the code needed would have been there at the beginning. Thus,

$$N_I(t_f) = S(1-p_1+p_2)D_0e^{-\beta_{1m}t_f} \tag{41}$$

The the additional defect density at time t_f can be obtained using this equation.

$$D_{add} = \frac{N(t_f) - N_I(t_f)}{(1-p_1+p_2)S} \tag{42}$$

The DEF is given by

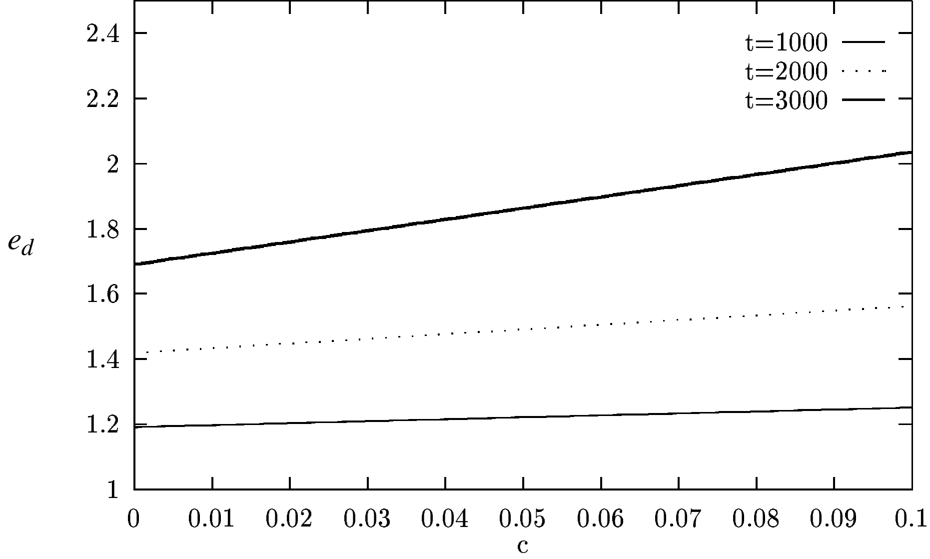

Figure 8. DEF due to code removal, plotted against c_1

$$e_d = \frac{S(1-p_1)D_0 e^{-\beta_1 t} + p_2 S D'_0 + N_{int}}{(1-p_1+p_2)SD_0} \quad (43)$$

Figure 9 shows a plot of e_d against t_1 for three pairs of values (p_1, p_2)=(0.1,0.1), (0.05,0.15), (0.15,0.05). As expected, the lower curve corresponds to the case when more code is deleted than added. However it can be observed that the tree curves are quite close together suggesting that for values assumed, interface defects inserted due to both added and removed code significantly affect the overall defect density. If a large part of a software module needs to be revised, in many cases it will be better to redo it from the beginning in order to avoid the interface defects. Note that because of significant number of interface defects, DEF is be larger than one even when the modification is made near the beginning.

4 Discussion

Above we have examined possible types of individual changes made at a time t_1. Generally in a project all these kinds of changes are made at different times. We would like to be able to combine these results to come up with a description of the overall process using a reasonably simple model. Some data is now available [12, 11] that give insight into typical process that might be encountered. Further investigations are needed to obtain a model that will represent requirement volatility as a factor in a multiplicative model for defect density [9, 3, 1]. It is easy to see from the results that the dependence on t_1 is exponential although in some special cases a liner approximation may be justified.

Often requirement volatility and hence changes in a the software are distributed over a period of time. It may be possible to lump the affect of such distributed events into one or more *equivalent events* for the ease of computation, a technique that is used in modeling solid-state silicon devices. The objective will be to obtain a model that is simple and still yields accurate

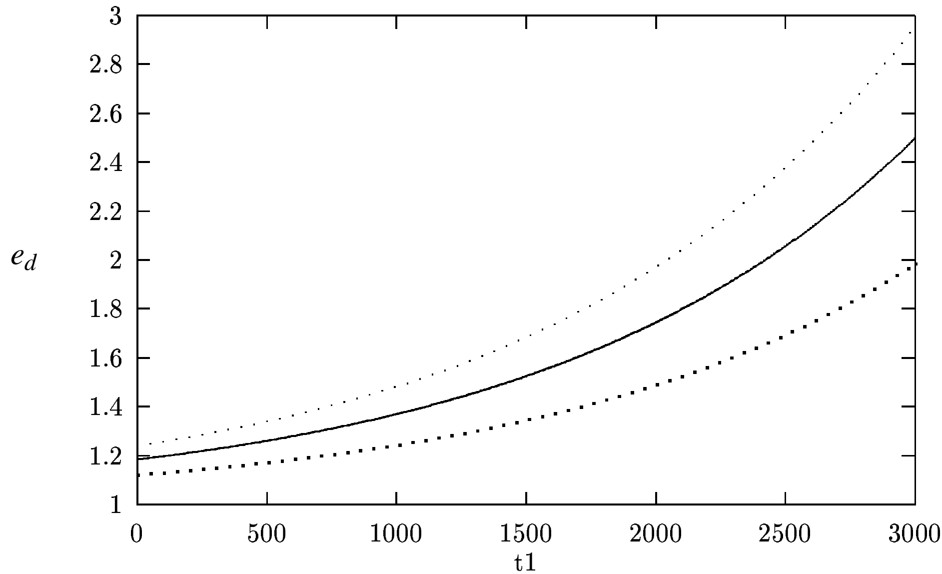

Figure 9. e_d **due to code modification, plotted against** t_1

estimates, perhaps after some calibration of parameters.

Here we have assumed that the development is in a test phase. Often some form of checking precedes such testing. For example, inspections can reveal defects early during the development process. It may be possible to regard such checking as testing, although clearly the values of the parameters will be different. It may be possible to reformulate the above results for application during the early phases when inspection and code walk-through are used.

5 Conclusions

Here we have analyzed the influence of changes in a program when testing has already been initiated. We have examined the effect of replacing a component with another component of the same size, as well as general cases when software is added, deleted and modified. All the results show that changes have more influence on defect density when they occur closer to the end of the testing effort. This temporal dependence is generally exponential. Changes made very early can be relatively inconsequential, but those occurring later can raise defect density quite significantly.

We have seen that in some cases, we must consider the interface defects to take into account the interaction among software blocks.

Further work is needed to come up with a general model that will relate a few measures that can be easily evaluated or estimated to the overall defect density. This will require a study of typical patterns of requirement changes over time.

6 Acknowledgement

The authors would like to thank John Musa for his suggestions about the influence of requirement volatility on defect density [6].

References

[1] J. A. Denton. *ROBUST, An Integrated Software Reliability Tool*. Colorado State University, 1997.

[2] J. Henry and S. Henry. Quantitative assessment of the software maintenance process and requirements volatility. In *Proc. of the ACM Conference on Computer Science*, pages 346–351, 1993.

[3] Y. K. Malaiya and J. A. Denton. What do software reliability parameters represent? In *Proc. International Symposium on Software Reliability Engineering*, pages 124–135, Albuquerque, NM, Nov. 1997.

[4] Y. K. Malaiya, A. von Mayrhauser, and P. Srimani. An examination of fault exposure ratio. *IEEE Transactions on Software Engineering*, pages 1087–1094, Nov. 1993.

[5] J. Musa. *Software Reliability Engineering*. McGraw-Hill, 1999.

[6] J. D. Musa. Personal communications. To Y. K. Malaiya, 1998.

[7] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability - Measurement, Prediction, Application*. McGraw-Hill, New York, 1987.

[8] M. Ohba and X. Chou. Does imperfect debugging affect software reliability growth? In *Proc. of the Int. Conference on Software Engineering*, pages 237–244, May 1989.

[9] Report. Methodology for software reliability prediction and assessment. Technical Report RL-TR-95-52, Vol. 1 and 2, Rome Labs, 1992.

[10] L. Rosenberg and L. Hyatt. Developing an effective metrics program. In *Proc. of the European Space Agency Software Assurance Symposium*, Mar. 1996.

[11] L. Rosenberg, L. Hyatt, T. Hammer, L. Huffman, and W. Wilson. Testing metrics for requirement quality. In *Proc. of the Int. Software Quality Week*, May 1998.

[12] G. Stark. Measurement to manage software maintenence. *Crosstalk*, July 1997.

[13] M. Takahashi and Y. Kamayachi. An empirical study of a model for program error prediction. In *Proc. of 8th International IEEE Conference on Software Engineering*, pages 330–336. IEEE, Aug. 1985.