Code Churn: A Measure for Estimating the Impact of Code Change

Sebastian G. Elbaum
John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
Phone: 885-4077/7789
{elbaum, jmunson}@cs.uidaho.edu

Abstract

This study presents a methodology that will produce a viable fault surrogate. The focus of the effort is on the precise measurement of software development process and product outcomes. Tools and processes for the static measurement of the source code have been installed and made operational in a large embedded software system. Source code measurements have been gathered unobtrusively for each build in the software evolution process. The measurements are synthesized to obtain the fault surrogate. The complexity of sequential builds is compared and a new measure, code churn, is calculated. This paper will demonstrate the effectiveness of code complexity churn by validating it against the testing problem reports.

1. Introduction

As software systems changes over time, it is very difficult to understand, measure and predict the effect of changes. We would like to be able to describe numerically how each system differs from its successor or predecessor.

Over a number of years worth of study, we can now establish the distinct relationship between software faults and certain aspects of software complexity [3,4]. When a software system is built for the first time, we have little or no direct information as to the location of faults in the code. In the absence of this information, we can use our relative complexity measure as a fault surrogate [12]. That is, if the relative complexity of a module is large, then it will likely have a large number of latent faults. If, on the other hand, the relative complexity of a module is small, then it will have fewer faults.

As the software system progresses through a number of sequential builds, faults will be identified and the code will be changed in an attempt to eliminate the identified faults. The introduction of new code, however, is just as fault prone a process as was the initial code generation. Faults will be injected during this evolutionary process.

Code does not always change in response to fixing faults. Some changes to code represent enhancements or changes in the code in response to evolving requirements. These incremental changes will also result in the injection of still more faults. As a system progresses through a series of builds, the relative complexity of each program module that has been altered will also change. Because of its role as a fault surrogate, the rate of change in relative complexity will serve as a good index of the rate of fault injection.

Software testing is supposed to aid in the detection and removal of software faults. The general notion of software test is that the rate of fault removal will generally exceed the rate of fault injection. In most cases, this is probably true. Some changes are rather more heroic than others. During these more substantive change cycles, it is quite possible that the actual number of faults in the system will rise.

We would be very mistaken, then, to assume that software test will monotonically reduce the number of faults in a system. This will only be the case when the rate of fault removal exceeds the rate of fault injection. The rate of fault removal is relatively easy to measure. The rate of fault injection is much more tenuous. It is directly related to the net change in the code from one sequential software build to the next. Thus, the first step in understanding the nature of the fault injection process involves the establishment of a methodology for measuring the nature of changes that occur in the build process. In other words, we wish to measure accurately the software evolution process, specifically as this measurement relates to the fault injection process.

2. Relative Complexity and a Measurement Baseline

The measurement of an evolving software system through the shifting sands of time is not an easy task. Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property that, when another point is chosen, the exact same picture of software evolution emerges, only the perspective changes [10]. The individual points involved in measuring software evolution are individual builds of the system.

A major problem emerges in the measurement of software systems in that the complexity metrics that we wish to use are all on different scales [8]. Comparing different modules within a software system by using these measurement data is complicated by this fact. Take for example the data in Table 1. This table provides the values for two metrics; statement count, *Stmnts*, and number of cycles in the control flowgraph, *Cycles*. These measurements are taken for two different builds of the system. It would be very difficult to assert that Module A is more complex than Module B on Build 1. Certainly, *Stmnts* is less than that for module B, but *Cycles* is greater. Now consider the same two modules for build 2. Has the system, as represented by these two modules, become more complex or less complex between these two builds? The total number of statements has decreased by ten, but the total number of cycles has increased by two. Again, it is difficult to assert that there has been an increase or decrease in overall complexity.

	Build 1		Build 2	
Module	A	B	A	B
Stmnts	200	250	210	230
Cycles	20	15	19	18

Table 1. A Measurement Example

In order to measure successive builds of a system, a referent system, or baseline, must be established. We will use the relative complexity metric in the establishment of a suitable baseline. We have developed the methodology for the relative complexity metric over a number of years. This metric is obtained in a two step process from a complexity metric suite. First, factor scores on each program module in the baseline, or referent program build, for each set of metrics, are created by principal components analysis together with an orthogonal transformation matrix. The orthogonal factor scores (domain metrics) are then combined into a single, relative complexity measure [3].

Next, it would be useful if each of the program modules in a software system could be characterized by a single value representing some cumulative measure of complexity. The objective in the selection of such a linear function, *g*, is that it be related in some manner to software faults either directly or inversely such that $g(x) = as + b$ where *x* is some unitary measure of program complexity. The more closely related *x* is to software faults, the more valuable the function, *g*, will be in the anticipation of software faults. Previous research has established that the relative complexity metric, ρ, has properties that will be useful in this regard [3, 5, 11].

The relative complexity, ρ, of the factored program modules may be represented as

$$\rho_i = \sum_j \lambda_i d_{ij}$$

where λ_j is the eigenvalue associated with the j^{th} factor and d_{ij} is the j^{th} domain metric of the i^{th} program module. Each of the eigenvalues represents the relative contribution of its associated domain to the total variance explained by all of the domains. In essence, then, the relative complexity metric is a weighted sum of the individual domain metrics. In this context, the relative complexity metric represents each raw complexity metric in proportion to the amount of unique variation contributed by that complexity metric. As we will see, the real utility of this measure from a software evolution perspective is that it will permit us to measure the amount of change in a system and also the rate of change in complexity of a software system.

Relative complexity, has been established as a successful surrogate measure of software faults [12]. It seems only reasonable that we should use it as the measure against which we compare different builds. Since relative complexity is a composite measure based on the raw measurements, it incorporates the information represented by *Statements*, *Cycles* and all the other raw metrics found to be related to software faults.

By definition, the average relative complexity, $\bar{\rho}^b$, of the baseline *system* will be

$$\bar{\rho}^b = \frac{1}{N^b}\sum_{i=1}^{N^b}\rho_i^b = 50,$$

where N^b is the cardinality of the set of modules on build *b*, the baseline build. Relative complexity for the baseline build is calculated from standardized values using the mean and standard deviation from the baseline metrics. The relative complexities are then scaled to have a mean of 50 and a standard deviation of 10. For that reason, the average relative complexity for the baseline system will always be a fixed point. Subsequent builds are standardized using the means and standard deviations of the metrics gathered from the baseline system to allow comparisons. The formula for calculating average relative complexity for subsequent builds is given as

$$\bar{\rho}^k = \frac{1}{N^k}\sum_{i=1}^{N^k}\rho_i^k ,$$

where N^k is the cardinality of the set of program modules in the k^{th} build and ρ_i^k is the relative complexity for the i^{th} module of that set.

The total relative complexity, *R*, of the system is simply the sum of all relative complexities of each module,

$$R = \sum_{i=1}^{N^b}\rho_i^b .$$

The principle behind relative complexity is that it serves as a fault surrogate. That is, it will vary in precisely the same manner as do software faults. The fault potential r_i of a particular module *i* is directly proportional its value of the relative complexity fault surrogate. Thus,

$$r_i = {\rho_i}/{R} .$$

Our ability to locate the remaining faults in a system will relate directly to our exposure to these faults [6,9]. If, for example, at the j^{th} build of a system there are g_i^j remaining faults in module *i*, we can not expect to identify any of these faults unless some test activity is allocated to exercising module *i*.

As the code is modified over time, faults will be found and fixed. However, new faults will be introduced into the code as a result of the change. In fact, this fault injection process is directly proportional to change in the program modules from one version to the next [10].

As a module is changed from one build to the next in response to evolving requirements changes and fault reports, its complexity will also change. Generally, the net effect of a change is that complexity will increase. Only rarely will its complexity decrease. It is now necessary to describe the measurement process for the rate of change in an evolving system.

3. Measuring Software Evolution

A software system consists of one or more software modules. As the system grows and modifications are made, the code is recompiled and a new version, or build, is created. Each build is constructed from a set of software modules. The new version may contain some of the same modules as the previous version, some entirely new modules and it may even omit some modules that were present in an earlier version. Of the modules that are common to both the old and new version, some may have undergone modification since the last build. When evaluating the change that occurs to the system between any two builds (software evolution), we are interested in three sets of modules. The first set, M_C, is the set of modules present in both builds of the system. These modules may have changed since the earlier version but were not removed. The second set, M_A, is the set of modules that were in the early build and were removed prior to the later build. The final set, M_B, is the set of modules that have been added to the system since the earlier build.

The relative complexity of the system R^i at build i, the early build, is given by

$$R^i = \sum_{c\in M_c}\rho_c^i + \sum_{a\in M_a}\rho_a^i .$$

Similarly, the relative complexity of the system R^j at build j, the later build is given by

$$R^j = \sum_{c\in M_c}\rho_c^j + \sum_{b\in M_b}\rho_b^j .$$

The later system build is said to be more complex if $R_i > R_j$.

As a system evolves through a series of builds, its complexity will change. This complexity is measured by a set of software metrics. One simple assessment of the size of a software system is the number of lines of code per module. However, using only one metric may neglect information about the other complexity attributes of the system, such as control flow and temporal complexity. By comparing successive builds on their domain metrics it is possible to see how these builds either increase or decrease based on particular attribute domains. Using relative complexity, the overall system complexity can be monitored as the system evolves.

Regardless of which metric is chosen, the goal is the same. We wish to assess how the system has changed, over time, with respect to that particular measurement. The concept of a code delta provides this information. A code delta is, as the name implies, the difference between two builds as to the relative complexity metric.

measure of code churn, χ, for module m is simply

$$\chi_m^{i,j} = \left|\delta_m^{i,j}\right| = \left|\rho_m^j - \rho_m^i\right|$$

The total change of the system is the sum of the code delta's for a system between two builds *i* and *j* is given by

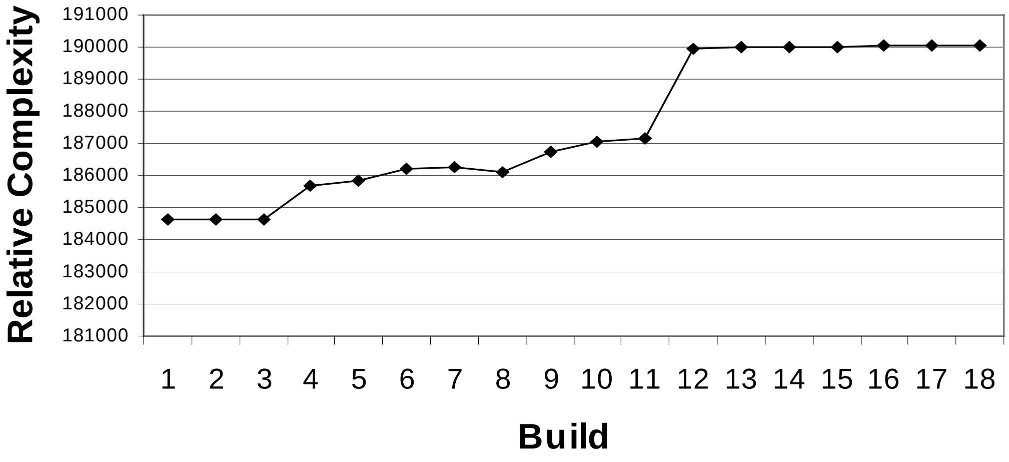

Figure 1. Software Evolution Over Successive Builds

$$\Delta^{i,j} = \sum_{c\in M_c} \delta_c^{i,j} - \sum_{a\in M_a} \rho_a^i + \sum_{b\in M_b} \rho_b^j \ .$$

Similarly, the net code churn of the same system over the same builds is

$$\nabla^{i,j} = \sum_{c\in M_c} \chi_c^{i,j} + \sum_{a\in M_a} \rho_a^i + \sum_{b\in M_b} \rho_b^j \ .$$

With a suitable baseline in place, and the module sets defined above, it is now possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the increase or decrease in system complexity as measured by a selected metric, code delta, or we can measure the total amount of change the system has undergone between builds, code churn.

A limitation of measuring code deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent complexity, the code delta for the system will be close to zero. The overall complexity of the system, based on the metric used to compute deltas, will not have changed much. However, the reliability of the system could have been severely effected by the process of replacing old modules with new ones. What we need is a measure

For purposes of demonstration, a large embedded real-time system, QTB, has been evaluated in depth. This is a real time system in charge of booting, monitoring and failure recovering for a hardware system. It has approximately 300K LOC in more than 3700 modules (functions) programmed in C. The overall trend in the relative complexity between nineteen successive builds is shown in Figure 1. The pattern shown here is quite typical of an evolving software system. In this case, the build labeled as Build 1 is not the initial build, but the one used as baseline. In looking at this figure we can see that there are periods of relative quiescence and also periods of great change in the system. The overall trend is always towards increased complexity.

The change in the relative complexity in a single module between two builds may be measured in one of two distinct ways. First, we may simply compute the simple difference in the module relative complexity between build *i* and build *j*. We have called this value the code delta for the *m* module, or $\delta_m^{i,j} = \rho_m^j - \rho_m^i$. The absolute value of the code delta is a measure of code churn. In the case of code churn, what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault insertion, removing a lot of code is probably as catastrophic as adding a bunch. The new to accompany code delta that indicates how much change has occurred. Code churn is a measurement, calculated in a similar manner to code delta that provides this information.

When several modules are replaced between builds by modules of roughly the same complexity, code delta will be approximately zero but code churn will be equal to the sum of the value of ρ for all of the modules, both inserted and deleted. Both the code delta and code churn for a particular metric are needed to assess the evolution of a system.

The net code delta values and the net code churn for the QTB system discussed earlier are shown in Figure 2. The data represented in figure two is stated also in Table 2. In this case, the code churn and code delta values are computed between sequential builds. We can see that the greatest change activity as measured by code churn occurred on builds 3, 7, 8, and 11. The net change effect on the system, as measured by code deltas, is somewhat different. The effect of the changes on build 7 reflect a reduction in the system overall complexity. On build 3, not all of the changes resulted in increased system complexity in that the code delta value is less than the code churn value.

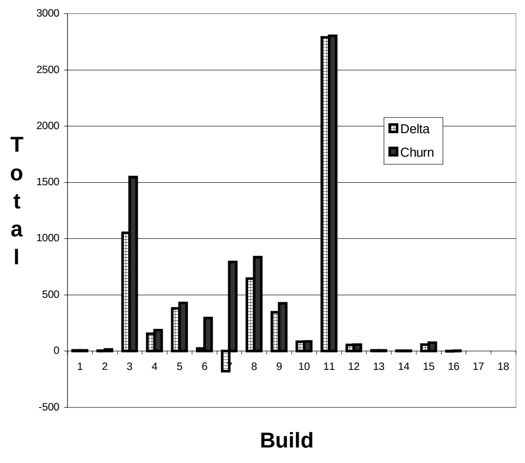

Figure 2. Software Evolution Over Successive Builds

Build 11 resulted in substantial change activity. In that code churn and code delta values are almost identical, we can see that the overwhelming majority of code constituted new, additional features to the code.

BUILD	RCM	DELTA	CHURN
1	184643.38	5.72	7.12
2	184648.81	5.43	16.78
3	185702.10	1053.28	1547.93
4	185857.61	155.50	187.11
5	186236.58	378.97	429.30
6	186261.81	25.23	294.18
7	186083.21	-178.60	791.87
8	186726.70	643.50	836.71
9	187072.37	345.65	426.24
10	187157.08	84.71	86.74
11	189945.91	2788.85	2804.02
12	190000.81	54.89	59.28
13	190007.10	6.29	6.38
14	190012.49	5.39	5.48
15	190069.36	56.87	75.08
16	190066.48	-2.88	3.86
17	190067.65	1.17	1.17
18	190067.81	0.16	0.21

Table 2: Evolution Data for QTB

4. Software Process

In software development, there are two key issues when trying to incorporate measurement and modeling into the process: to have a clear objective and to implement it with transparency. One of the fundamental convictions in our approach to software measurement is that there is a reason for doing it. We use measurement to determine the impact of a software change in the reliability of the system and to determine that existence of software faults. Transparency is a key issue when incorporating measurement in the development process because of the resistance that developers have in using them consistently. If measurement is not transparent, then it doesn't happen. Software is a leaving creature that changes as we are attempting to measure it, for that reason is extremely important to perform measurements consistently and precisely. The quality of any model will be conditioned by the quality of the measurements that are taken [2].

For the past two years, we have developed and refined a family of measurement tools that allow us to gather all the information automatically and independently of the developers. The first of those tools is called CMA (C Metric Analyzer). CMA generates a working set of complexity metrics that are known to be highly correlated to software faults. The driving forces in the development of this measurement tool have been to construct a suitable surrogate measure for software faults and to be able to plug-in the development process as a part of the configuration control system already in place. Each of the metrics included in the tool explains a significant amount of variation in the concomitant criterion measure of software faults. CMA version 1.0 measures the following fourteen metrics [7] listed in the table below.

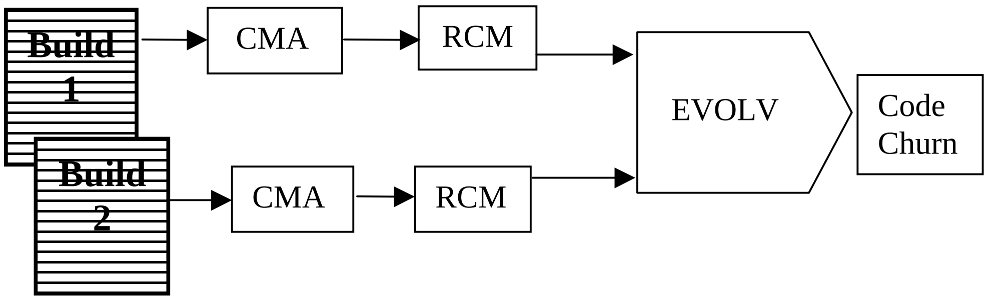

Figure 3. Software Tools and Processes

The second tool is PCA/RCM (Principal Components Analysis / Relative Complexity Metric). This tool performs the principal components analysis on a baseline build and stores the information that is needed to compute the relative complexity metric for successive builds.

Comm	Total comment count
ExStmt	Executable statements
NonEx	Non executable statements
N_1	Total number of operands
η_1	Unique operands
N_2	Total number of operators
η_2	Unique operators
η_3	Unique operators with overloading
Nodes	Number of nodes in the module control flowgraph
Edges	Number of edges in the module control flowgraph
Paths	Number of distinct paths in the module control flowgraph
MaxPath	Maximum path length in the module control flowgraph
$\overline{Path}$	Average path length in the module control flowgraph
Cycles	Total cycle count in the module control flowgraph

Table 3. Metric Definitions

The third tool is the software evolution tool evolution tool, EVOLV. This tool allows the automatic comparison of the complexities of two builds. EVOLV uses the relative complexity values stored by PCA/RCM for each one of the software modules of each build. It seeks to match equivalent modules for both builds. It takes into consideration what modules were added, removed or modified and it computes the code delta and code churn for the latest build.

These tools were incorporated into the development process. As it can be seen in Figure 3, the tools are part of a flow of metric information that occurs transparently and automatically. After all the software change requests intended to be in a build are completed, the build processes starts. At that exact time, a script launches the measurement tool over the modules that have been modified and stores the data. When the measurement is complete, PCA/RCM computes relative complexity for all the modules that have changed. At last, the script locates the previous build and calls the evolution tool. The evolution tool searches in the measurement files for the relative complexity value for every module involved in the builds and then computes the code delta and churn.

5. Results

We are lead to believe, based on our prior experiences, that code churn would be a very effective fault surrogate. To put it in the proper context, we want to evaluate some process measures and compare them with code churn. To this end, we designed and conducted a simple evaluation experiment [1]. The experiment was conducted in an industrial environment using all the software evolution data collected in the last six months for the QTB embedded real-time software system. During these six months interval there were a total of 19 builds. For each build, the number of software change requests that were addressed and incorporated into each one of those builds and the number of programmers directly involved in each one of them was recorded. A software change request is a document filled by a developer to solicit a change to a piece of software. This document is methodically reviewed and, if approved, allows the programmer to start with the modification of the code. Also during this same interval, the problem reports generated by regression testing were gathered. The problem reports are normally analyzed by a development team to evaluate their impact. Some problem reports might require an immediate software change request while others are assigned lower priority and are addressed in a later build. These trouble reports would serve as the criterion measure to assess the effectiveness of the evolution data through the computation of multiple R in sequential univariate regression models.

The data collected are presented in Table 3. This table is organized into six columns containing the software product metrics (code churn and code delta) and also the software process metrics. The software process metrics included in Table 3 are the change requests that were included in each build, the number of programmers that were directly involved in the process of code modification and the total number of problem reports that were completed by the regression testing engineers as the code was tested.

Build	Delta	Churn	Change Requests	People Involved	Problem Reports
1	5.72	7.12	5	4	3
2	5.43	16.78	8	10	1
3	1053.28	1547.93	5	2	6
4	155.5	187.11	16	5	1
5	378.97	429.30	6	5	4
6	25.23	294.18	7	4	5
7	-178.60	791.87	2	2	7
8	643.50	836.71	24	10	3
9	345.65	426.24	10	6	4
10	84.71	86.74	7	5	1
11	2788.85	2804.02	8	5	6
12	54.89	59.28	3	3	2
13	6.29	6.38	2	2	4
14	5.39	5.48	7	6	1
15	56.87	75.08	5	2	3
16	-2.88	3.86	3	3	3
17	1.17	1.17	1	1	3
18	0.16	0.21	1	1	0

Table 3. Experimental Outcome

We used a regression model sans constant term to calculate the Multiple R with each of the four measures with *problem reports* as the criterion variable in each model. The results of these regression studies are shown in Table 4. The model with the greatest Multiple R was the one with code churn. This model had a Multiple R of 0.728. It means that code churn is a powerful surrogate for software faults, accounting for more than 50% of the total variance in the fault measure as indicated by the R Square value from Table 4. When comparing code churn with some other predictors, following the same procedure, the Multiple R value for code churn were better as it is shown in Table 4 (the surrogates are ranked based on their predictive value).

Process Measures	Multiple R	R Squared	Equation	
			Coeff.	Std. Error
Code Churn	0.728	0.529	0.003	0.001
People	0.667	0.445	0.5	0.132
Change Requests	0.619	0.383	0.265	0.079
Code Delta	0.560	0.314	0.003	0.001

Table 4. Regression Analysis

The Pearson product moment correlations among the experimental variables are shown in Table 5. There are several interesting observations that may be made here. First, note the relationship between churn and delta. Had all changes resulted in new code, the correlation here would have been 1.0. The reported correlation of 0.940 is very close to 1.0. This means that the majority of changes made over the 19 builds resulted in increased code complexity. On the other hand, if this value had been –1.0, we would conclude that all changes made would have reduced the complexity of the total system.

	Delta	Churn	Change	People
Churn	0.940			
Change	0.260	0.236		
People	0.167	0.114	0.759	
Trouble	0.445	0.648	-0.014	-0.094

Table 5. Product Moment Correlations

Of the three independent variables of churn, deltas, and people measures, we can see that code churn has the greatest correlation with the criterion measure of trouble reports. Also of interest is the relationship between code churn and the people measure. This correlation is distinctly low. The number of software developers involved in changes is apparently not related to the magnitude of the change. Similarly, too many cooks apparently do not spoil the stew. There is no apparent relationship between the number of developers implementing a change and the number of trouble reports that resulted from the change.

This experiment has two distinct sources of noise in the primary criterion measure of regression problem reports. First, we are using the problem reports as actually report failure events. These failure events may, in fact, be the result of multiple faults. Second, the reports only correspond to the subset of the faults found in regression testing. Faults removed during code inspection and unit testing were not recorded. However, in the presence of these known and uncontrolled sources of variation in the criterion measure, the signal component is very strong. Code churn is a valuable early indicator of software faults. It will serve as a very good fault surrogate.

6. Summary

Complexity metrics can provide substantial information on the distinguishing differences among the modules of a software system in regards to the conduct of the testing process. There is reasonable evidence to support the conclusion that computer software metrics may be used as leading indicators of software quality in the test process. Further, these metrics can be used to guide (1) the formulation of a test plan for structuring the test process and (2) to allocate test resources to yield a maximum exposure to software faults during the test process.

The success of this technique is predicated on the ability to identify a surrogate for software faults. Relative complexity, and its build specific derivatives of code churn and code delta, is a stand-in for aspects of software quality that we cannot directly measure, such as software faults. From the results of our recent research, we believe that these derived complexity measures are stable and reasonable measures to guide the testing process. Unlike other metrics, the relative complexity metric combines, simultaneously, all attribute dimensions of all complexity metrics.

We have established that software complexity metrics, and subsequently the relative complexity metric together with code churn and code deltas, are closely associated with measures of program quality. This relationship may be exploited in the software development process as an aid to the inspection process and also in the testing process as an indicator of potential problem areas for the regression testing process.

References

[1] N. Fenton, S. L. Pfleeger and R. Glass, "Science and Substance, A Challenge to Software Engineers", IEEE Software, July 1994, pp.86-95.

[2] F. Lanubile, "Why Software Reliability Predictions Fail", IEEE Software, July 1996, pp.131-132,137.

[3] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications", IEEE Software, January 1996, pp.65-71.

[4] T. M. Khoshgoftaar and J. C. Munson "A Measure of Software System Complexity and Its Relationship to Faults," In *Proceedings of the 1992 International Simulation Technology Conference*, The Society for Computer Simulation, San Diego, CA, 1992, pp. 267-272.

[5] T. M. Khoshgoftaar and J. C. Munson , "Predicting Software Development Errors Using Complexity Metrics," *IEEE Journal on Selected Areas in Communications* 8, 1990, pp. 253-261.

[6] J. C. Munson, S. G. Elbaum, and R. M. Karcich, "Software Risk Assessment Through Software Measurement and Modeling", will appear *Proceedings of the 1998 IEEE Aerospace Applications Conference,* IEEE Computer Society Press.

[7] J. C. Munson and S. G. Elbaum, "A Standard for the Measurement of of C Complexity Attributes", Software Engineering Testing Lab, Technical Report #398.

[8] J. C. Munson, "Software Measurement: Problems and Practice," *Annals of Software Engineering,* J. C. Baltzer AG, Amsterdam 1995.

[9] J. C. Munson, "Software Faults, Software Failures, and Software Reliability Modeling", *Information and Software Technology*, December, 1996.

[10] J. C. Munson and D. S. Werries, "Measuring Software Evolution," *Proceedings of the 1996 IEEE International Software Metrics Symposium* , IEEE Computer Society Press, pp. 41-51.

[11] J. C. Munson and T. M. Khoshgoftaar "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, SE-18, No. 5, 1992, pp. 423-433.

[12] J. C. Munson and T. M. Khoshgoftaar, "The Relative Software Complexity Metric: A Validation Study," In *Proceedings of the Software Engineering 1990 Conference*, Cambridge University Press, Cambridge, UK, 1990, pp. 89-102.