UNDERSTANDING SOFTWARE FAULTS AND THEIR ROLE IN SOFTWARE RELIABILITY MODELING

525-61

44838

John C. Munson
Department of Computer Science
University of West Florida

This study is a direct result of an on-going project to model the reliability of a large real-time control avionics system. In previous modeling efforts with this system, hardware reliability models were applied in modeling the reliability behavior of this system. In an attempt to enhance the performance of the adapted reliability models, certain software attributes were introduced in these models to control for differences between programs and also sequential executions of the same program [8]. As the basic nature of the software attributes that affect software reliability become better understood in the modeling process, this information begins to have important implications on the software development process.

A significant problem arises when raw attribute measures are to be used in statistical models as predictors, for example, of measures of software quality. This is because many of the metrics are highly correlated [1, 4]. Consider the two attributes: lines of code, *LOC*, and number of program statements, *Stmts*. In this case, it is quite obvious that a program with a high value of *LOC* probably will also have a relatively high value of *Stmts*. In the case of low level languages, such as assembly language programs, there might be a one-to-one relationship between the statement count and the lines of code. When there is a complete absence of linear relationship among the metrics, they are said to be orthogonal or uncorrelated. Usually the lack of orthogonality is not serious enough to affect a statistical analysis. However, for the purposes of some statistical analysises such as multiple regression, the software metrics are so strongly interrelated that the regression results may be ambiguous and possibly even misleading. Typically, it is difficult to estimate the unique effects of individual software metrics in the regression equation. The estimated values of the coefficients are very sensitive to slight changes in the data and to the addition or deletion of variables in the regression equation.

Since most of the existing metrics have common elements and are linear combinations of these common elements, it seems reasonable to investigate the structure of the underlying common factors or components that make up the raw metrics. The technique we have chosen to use to explore this structure is a procedure called principal components analysis. Principal components analysis is a decomposition technique that may be used to detect and analyze collinearity in software metrics. When confronted with a large number of metrics measuring a single construct, it may be desirable to represent the set by some smaller number of variables that convey all, or most, of the information in the original set. Principal components are linear transformations of a set of random variables that summarize the information contained in the variables. The transformations are chosen so that the first component accounts for the maximal amount of variation of the measures of any possible linear transform; the second component accounts for the maximal amount of residual variation; and so on. The principal components are constructed so that they represent transformed scores on dimensions that are orthogonal [4].

Through the use of principal components analysis, it is possible to have a set of highly related software attributes mapped into a small number of uncorrelated attribute domains. This definitively solves the problem of multicollinearity in subsequent regression analysis [3]. There are many software metrics in the literature, but principal component analysis reveals that there are few distinct sources of variation, i.e. dimensions, in this set of metrics. It would appear perfectly reasonable to characterize the measurable attributes of a program with a simple function of a small number of orthogonal metrics each of which represents a distinct software attribute domain.

SOFTWARE ATTRIBUTES AND SOFTWARE FAULTS

There is now sufficient evidence to support the conclusion that there is a distinct relationship between software faults and measurable program attributes and that this information will yield specific guidelines for the design of reliable software. In particular, software complexity measures are distinct program attributes that have this property [3]. If a program module is measured and found to be complex, then it will have a large number of faults. These faults may be detected by analytical methods, e.g., program inspections. The faults may also be identified based on the *failures* that they induce when the program is executing. A program may preserve a number of latent faults over its lifetime in that the particular manner that it is used may never cause the complex code sequences to execute and thus never expose the faults. Alternatively, a program may be forced to execute its complex code segments early in its life cycle and thus fail frequently early on followed by reliable service.

Code faults are not inserted by some random process analogous to the assignment of chocolate chips to cookies in the cookie manufacturing business. Faults occur in direct relationship to the complexity of the programming task. It is said about one of our recent presidents that he could not walk and chew gum at the same time. A programmer faced with the task of converting a complex requirement into a complex algorithm in a rich programming language has a much more difficult task than just walking and chewing. It is quite reasonable to expect that the programmer will make errors. These errors will express themselves as faults in the program. From a maintenance perspective, it will be very expensive and time consuming to find and to fix these faults. The real problem is to identify design rules that will restrict code faults from being introduced in the first place.

The concept of module granularity is relevant in terms of our description of a software system. At the lowest level of granularity, the micro level, we might define a module to be the smallest compilable unit of code. The next level of module granularity might be called the top level module granularity. In this case, all `include` compiler directives have been preprocessed by the compiler system. Typically, these top level modules will be the smallest units that may be measured by an execution profiler, a concept that will be necessary later in this discussion. Yet another level of granularity would be the functional subsystem level, i.e. the collection of modules grouped together in such a subsystem that would work in concert to solve a particular problem.

Whole classes of faults can be characterized as within-module faults. An example of this might be the case where a local identifier within a module is inadvertently assigned the wrong value by a programmer. One stage of fault analysis will examine the relationship of micro complexity measures such as the Halstead software science measures to faults occurring within module boundaries.

At the macro complexity level, we are interested in software measures that describe the interaction of program modules. This would include the macro measures of software coupling, call graph features, or more global software system metrics such as the aggregate complexity metric [2]. At this level we are interested in the relationship between macro complexity measures such a coupling complexity metrics and faults that are found on the inter-module interface level.

Yet another class of software faults do not relate to physical problems in the code but to temporal sequencing anomalies in a real-time control environment. This fault class will represent the highest level granularity. This is the operating system task management level. In this case the operating system or operating environment is responsible for the sequencing of the operation of program modules and the successful rendezvous of these modules.

A DOMAIN MODEL OF SOFTWARE ATTRIBUTES

In most linear modeling applications with software metrics, such as regression analysis and discriminant analysis, the independent variables, or metrics, are assumed to represent some distinct aspect of variability not clearly present in other measures. In software development applications, the independent variables, in this case, the complexity metrics, are strongly interrelated or demonstrate a high degree of multicollinearity. In cases like this, it will be almost impossible to establish the unique contribution of each metric to the model. One distinct result of multicollinearity in the independent measures is that the regression models developed using independent variables with a high degree of multicollinearity have highly unstable regression coefficients. Such models may be subject to dramatic changes due to additions or deletions of variables or even discrete changes in metric values. To circumvent this problem, principal components analysis has been used, quite successfully, to map the metrics into orthogonal attribute domains [4]. Each principal component extracted by this procedure may be seen to represent an underlying common attribute domain.

To aid in the visualization of this concept, the principal components analysis of a large avionics software system will be examined. In this specific example, a set of 13 metrics were identified. These metrics were chosen because they were thought to represent a number of distinct software complexity attributes. Each of the 13 measurements were obtained from a set of 1300 program modules that comprise the avionics system.

Many of the 13 metrics are known to be highly correlated. The first step in the investigative process is to determine how many distinct orthogonal attributes are really being measured by these 13 raw complexity metrics. Principal components analysis will reveal this orthogonal attribute structure. From Table 1, which contains the results of the principal components analysis of the 1300 software modules, we can see that most of the information represented by the original 13 metrics can be reduced to three orthogonal attribute domains. This table contains the values that assess the strength of the relationship between each raw metric and the three resulting orthogonal domains. There are a group of six metrics, for example, that are closely related to the first domain in the new orthogonal attribute domain structure. These are highlighted with boldface type. Similarly there are four raw metrics associated with the second attribute and three with the third. In some cases it is useful to associate names with the domains. The metrics most closely associated with **Domain1** in Table 2 are those whose table values are the largest in column one of the table. For clarity, they have been typed in bold. All of these metrics seem to share a common property of **Size** measurement. Similarly, the metrics associated with **Domain2** seem to share a **Structure** measurement and those in **Domain3** a **Control** measure.

If we carefully examine the set of 13 measures used above we can observe that there are some conceptual areas of software attributes that are not represented in this set of metrics. For example, there is the matter of data structure complexity. Clearly, there are no measures present in the set for this software attribute. The problem is that we do not know just how many distinct, measurable attributes a software system might have. But we do know that these 13 metrics are only measuring three distinct, uncorrelated attributes.

Attribute Domain Model

The objective now is to begin to build and extend a model for software attributes. This model will contain a set of orthogonal attribute domains. Once we have such a model in place we would then like to identify and select from the attribute domain model those attributes that are correlated with a software quality measure, such as number of faults. Each of the orthogonal attributes will have an associated metric value that is uncorrelated with any other attribute metrics. Each of these attributes may potentially serve to describe some aspect of variability in the behavior of the software faults in a program module.

One the primary purposes in the use of principal components analysis is to reduce the dimensionality of the attribute problem. In the specific case of the avionics system under current scrutiny, there were initially thirteen apparent software attributes that were being measured. Through the use of principal components analysis, we find that there are three distinct attribute domains that will serve to explain most of the variation observed in the original set of thirteen metrics. Further, we can transform the thirteen raw attribute measures for each of the 1300 program modules into three new uncorrelated metrics, one for each of the new orthogonal attribute domains. While this reduction in the number of metrics has simplified the problem somewhat, we really would like to represent each program with a single metric that would allow us to compare (order) each of the program modules in comparison to the rest of the program modules.

There have been some tragically ill-considered attempts to design software systems reflecting the complexity of the object being designed. The most notable of these attempts relates to the use of McCabe's measure of cyclomatic complexity V(g). For some unknown reason, magic values of cyclomatic complexity are now being incorporated into the requirements specifications of some software systems. For example, we might choose to specify that no program module in the software system should have a cyclomatic complexity greater than an arbitrary value of, say 15. This is a very good example of how software measures might well be used in the design process.

There are potentially catastrophic consequences associated with this univariate design criterion. First, there is little or no empirical evidence to suggest that a module whose cyclomatic complexity is greater than 15 is materially worse than one whose cyclomatic complexity is 14. Secondly, and most important is the fact that if, in the process of designing a software module, we find that the module has a cyclomatic complexity greater than 15, the most obvious and common solution to the problem is to divide the software module into two distinct modules. Now we will certainly have two modules whose cyclomatic complexity is less than 15. The difficulty here, is that instead of one program module, we have created two, or possibly three, in its place. This will increase the macro complexity of measures related to complexity. In other words, we have decreased cyclomatic complexity, but we have increased coupling complexity. The result of the ignorant decision may well be that the total *system* complexity will increase. This, in turn will lead to a concomitant increase in total faults.

From the attribute domain model, we can see that there are many distinct software complexity domains. If we make design decisions on the basis of incomplete measurements, we run the risk of doing ourselves real damage. As above, we may make a design decision to reduce a measurement in one domain but this may in fact cause a concomitant increase in measures in other complexity domains. The net result of this univariate decision is that the net complexity of a system may rise. While it is true that we may have fewer faults associated with aspects of control complexity, we run the risk of increasing the count of faults associated with coupling considerations.

In order to simplify the structure of software complexity even further than the orthogonal domains produced by the principal components analysis 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 function, g, is that it be related in some manner to software faults either directly or inversely such that $g(x) = ax + b$ 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 might be useful in this regard. The relative complexity metric, ρ, is a weighted sum of a set of uncorrelated attribute domain metrics [6, 7]. This relative complexity metric represents each raw metric in proportion to the amount of unique variation contributed by that metric.

A sample of the relative complexity values for the avionics software system is shown in Table 3, together with the domain metrics. This table contains three compartments. The first compartment shows the program modules of the least complexity; the second compartment shows those of average relative complexity; and the third compartment contains the complexity values from the most complex modules (by relative complexity). The last column in the table lists the total faults found in each of the associated modules. We can see an obvious relationship between the relative complexity of a program module and the number of faults found in it. The bivariate correlation between these two variables is approximately 0.80 for all of the data from the 1300 program modules.

Maintaining a Software Measurement Data Base

From the standpoint of design methodology, we would like to be able to use the measurements from past software development efforts for the purposes of developing a preliminary assessment of current or active software design project. In other words, we would like to have the ability to use a past system to serve as a baseline for a current development project. In essence the objective is to use an existing data base to baseline subsequent measures from a software system currently in development. We might choose, for example, to take the first build of a real-time control software system that was developed in the past and use this for a real-time control software system currently being developed. In this sense, all subsequent software measures on new systems will be transformed relative to the baseline system.

The ability to use information from past development projects in current design work is most important. This is due to the fact that many of the software quality and reliability attributes of a system can only be measured after the system has been in service for some time. If a software system under current development is directly comparable to one that had demonstrated quality and/or reliability problems in the past, there is evidence to suggest that the design had better be modified, and soon.

The attribute measures presented so far are static measures of the program. They measure such features of the program as its size and the complexity of its control structure. If the functionality of a program was extremely restricted, these static measure might well be sufficient to describe the program entirely. Most modern software systems, however, have a broad range of functionality. Consider, for example, the software system for a typical spreadsheet program. The number of distinct functions in such a system and the number of ways that these function might be exercised are both very large numbers. In addition to static measures of program attributes, we must also be concerned with dynamic measures of programs as well.

Execution Profile

A software system is written to fulfill a set of functional requirements. It is designed in such a manner that each of these functionalities is expressed in some code component. In some cases there is a direct correspondence between a particular program module and a particular functionality. That is, if the program is expressing that functionality, it will execute exclusively in the module in question. In most cases, however, there will not be this distinct traceablility of functionality to modules. The functionality will be expressed in many different code modules. In addition to the user profile derived from the customer profile, one can derive a functional profile for each system mode of operation. Of course, each mode of operation has its own occurrence probability. For a given system mode, it is possible to refine that mode into the functions it requires and each function's occurrence probability providing a quantitative distribution of the relative use of various functions.

As a program is exercising any one of its many functionalities, it will apportion its time across one to many program modules. This temporal distribution of processing time is represented by the concept of the execution profile. In other words, if we have a program structured into n distinct modules, the execution profile for a given functionality will be the proportion of time spent in each program module during the time that the function was being expressed.

Another way to look at the execution profile is that it represents the probability, p_i, of execution occurring in module m_i at any point in time. When a software system is running a fixed function there is an execution profile for the system represented by the probabilities, $p_1, p_2, p_3, \cdots, p_n$. For our purposes, p_i represents the probability that the i^{th} module, in a set of n modules, is in execution at any arbitrary time.

Each functionality will have is own, possibly unique, execution profile. The specific implications for software design in the concept of the execution profile may be drawn from information theory. In this case, each scenario under which the program may execute has its own set of execution profiles. Each of the execution profiles may be assessed in terms of its entropy defined as follows:

$$h = -\sum_{i=1}^{n} p_i \log_2 p_i .$$

The point of maximum entropy will occur when all modules will execute with an equal probability. In which case,

$$h_{\max} = \log_2 n - 1.$$

This point of maximum entropy would occur in a circumstance where all modules of a program were executed for precisely the same amount of time under a given input scenario. It is a point of maximum surprise. There is no information as to where the program is likely to be executing at any point in time. At the other extreme, the point of minimum entropy, 0, is the point at which the program will execute in only one of its modules. The probability of executing occurring in this module is 1.0. The probability of it executing any other module is 0.

The concept of entropy may also be used to evaluate a design in terms of its modularity. We can see that the entropy for a set of n program modules is bounded above by $\log_2 n - 1$. From the standpoint of entropy, the maximum effect of a decision to increase the number of modules in a design from n to $n+1$ is basically the difference between $\log_2 n$ and $\log_2 (n+1)$. For small n, this difference might be substantial. For large n, the incremental effect of the additional module is not that great.

The entropy of a design is not a sufficient condition in its own right for the evaluation of a design. There is clearly an interaction between entropy and complexity. A design of low entropy implies that a great deal of time is spent in relatively few modules. If the associated modules have small values of relative complexity then its reliability should be high. If, on the other hand, the most frequently executed modules are also very complex, we would anticipate a low reliability for the system.

Functional Complexity

As per the earlier discussion of relative complexity, it can be seen that it is possible to characterize a program module by a single value such as relative complexity. This relative complexity measure, ρ, of a program is a measure of the program at rest. When a program is executing, not every one of its modules will receive the same level of exposure. This is evident in the concept of execution profile. Some executions might result in very complex program modules being executed. Other program input stimuli may cause the program to execute only its least complex modules.

There is, then, a dynamic aspect to program complexity that is related to its entropy under a particular test scenario. The functional complexity, ϕ, of the system running an application with an execution profile defined above is then:

$$\phi = \sum_{j=1}^{n} p_j \rho_j$$

where ρ_i is the relative complexity of the j^{th} program module and p_j is the execution probability of this module. This is simply the expected value of relative complexity under a particular execution profile. The execution profile for a program can be expected to change across the set of program functionalities. In other words, for each functionality, f_i, there is an execution profile represented by the probabilities $p_1^i, p_2^i, \ldots, p_n^i$. As a consequence, there will be a functional complexity ϕ_i for each function, f_i execution, where

$$\phi_i = \sum_{j=1}^{n} p_j^i \rho_j$$

This is distinctly the case during the test phase when the program is subjected to numerous test suites to exercise differing aspects of its functionality. The functional complexity of a system will vary greatly as a result of the execution of these different test suites.

Given the relationship between complexity and embedded faults, we would expect the failure intensity to rise as the functional complexity increases. If an application is chosen in such a manner that there are high execution probabilities for the complex modules, then the functional complexity will be large and the likelihood of a failure event during this interval would be relatively high. Given the relationship between software complexity and software faults, some applications will lead to consistent failures while others will not. The most important message, here, is that a software system will fail in direct relationship its functionality. This is a very predictable and understandable relationship.

SOFTWARE RELIABILITY MODELING

Almost all of the current software reliability modeling approaches are simply derivatives of hardware reliability models. Our view of complex software systems is colored by our experience with complex mechanical or electronic systems. In the case of software systems it is evident that these systems are clearly a different animal. Consider the case of a jet engine as a mechanical system. When this system is operating, essentially all of its components are operating together. An operational test of this system may be achieved by simply breathing life into it. This is not the case with software systems. These systems contain many independent components, or modules. Taken in their totality there are potentially a vast number of distinct paths that may be taken through these systems as a function of the different inputs to the system. Hence, the dynamic complexity of the software system will depend on the inputs to the system. The net effect of differing inputs to the system is that the operational or functional complexity of the system will change in response to the varying inputs. Given the association between module complexity and errors, it follows that as applications change over time intervals, so too, will the likelihood of faults change with respect to time.

If we closely examine the attempts to fit simple Non-homogeous Poisson Process models or simple two parameter Weibull type models to empirical data, a consistent pattern emerges. There are systematic departures in the observed values from the predicted values. These departures represent significant trends in the residuals. There is something systematically happening in the execution of the software. This clearly does not represent noise in the system. It is the thesis of this paper that the departure of the residuals from the random pattern that we would expect due to measurement errors and other sources of noise is directly attributable to the functional complexity of the software in execution.

In order to demonstrate the integration of functional complexity into the modeling process, the Weibull distribution will be used. This is basically a three parameter distribution whose cumulative distribution for a random variable, x, is given as follows:

$$F(x;\alpha,\beta,\gamma) = 1-e^{\left[\frac{x-\gamma}{\alpha-\gamma}\right]^{\beta}}$$

where $\beta > 0$, $\alpha > 0$ and $\gamma \geq 0$. The location parameter, γ, will represent the minimum life of the system. The parameter, β, is the shape parameter or Weibull slope. The parameter, α, is the scale parameter.

In the particular case where we wish to represent the cumulative failure probability of a system with a minimum life of zero, the γ term vanishes yielding a two parameter Weibull distribution,

$$F(x;\alpha,\beta) = 1-e^{\left[\frac{x}{\alpha}\right]^{\beta}} .$$

Let us now surmise that the scale of the distribution represents the functional complexity of the software varying by functionality over time. The former parameter, α, will now be derived from empirical data. This form of the Weibull distribution is a one parameter distribution in β. The bottom line of this new modeling process is the observation that the probability of the failure of a software system is not just strictly a function of time. It also appears to be a function of the actual code that is executing at any point in time. This new process may be summarized quite succinctly as follows: Tell me what the software will be executing and I will tell you what its reliability will be for that application.

SUMMARY

It is increasingly evident that the reliability of a software system is largely determined during program design. One distinct aspect of the software design process that lends itself to measurement is the decomposition of the functionality of a system into modules and the subsequent interaction of the modules under a given set of inputs to the software. The actual distribution of execution time to each software module for a given input scenario is recorded in the execution profile for the system. For a given input scenario to a program, the execution profile is a function of how the design of the software has assigned functionality to program modules.

The reliability of a software system may be characterized in terms of the individual software modules that make up the system. From the standpoint of the logical description of the system, these functional components of the larger system are, in fact, operating in series. If any one of these components fails, the entire system will fail. The likelihood that a component will fail is directly related to the complexity of that module. If it is very complex the fault probability is also large. The system will be as reliable as its weakest component.

It is now quite clear that the architecture of a program will be a large determinant of its reliability. Some activities that a program will be asked to perform are quite simple. They will be easily understood by designers and programmers alike. The resulting code will not likely contain faults. If, on the other hand, the specified functionality of a program is very complex and, as a result ambiguous, then there is a good likelihood that this functionality will be quite fragile due to the faults introduced through the very human processes of its creation. A more realistic approach to software reliability modeling will reflect the software reliability in terms of the functionality of the software.

REFERENCES

[1] V. Cote et al., "Software Metrics: An Overview of Recent Results," *Journal of Systems and Software*, **8**, 1988, pp. 121-131.

[2] T. M. Khoshgoftaar and J. C. Munson, "A Measure of Software System Complexity and Its Relationship to Faults," *Proceedings of the 1992 International Simulation Technology Conference* Houston TX November, 1992.

[3] T. M. Khoshgoftaar and J. C. Munson, "Predicting Software Development Errors Using Complexity Metrics," *IEEE Journal on Selected Areas in Communications* Vol. 8, No. 2, February 1990, pp. 253-261.

[4] J. C. Munson and T. M. Khoshgoftaar, "The Dimensionality of Program Complexity," *Proceedings of the 11th Annual International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 245-253.

[5] J. C. Munson and T. M. Khoshgoftaar, "Regression Modeling of Software Quality: An Empirical Investigation," *Journal of Information and Software Technology*, Vol. 32 No. 2 March 1990, pp. 105-114.

[6] J. C. Munson and T. M. Khoshgoftaar, "Applications of a Relative Complexity Metric for Software Project Management," *Journal of Systems and Software*, Vol. 12, No. 3, 1990, pp. 283-291.

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

[8] J. C. Munson and T. M. Khoshgoftaar, "The Use of Software Complexity Metrics in Software Reliability Modeling," *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, May, 1991, pp. 2-11.

	Domain1 Size	**Domain2 Structure**	**Domain3 Control**
N_2	**0.964**	0.106	0.096
LOC	**0.950**	0.119	0.166
N_1	**0.946**	0.117	0.064
η_2	**0.921**	0.029	0.057
Stmts	**0.740**	0.225	0.472
Comm	**0.662**	0.056	0.395
Max–Path	0.109	**0.914**	0.303
Path	0.115	**0.909**	0.296
Paths	0.128	**0.853**	0.189
Cycles	0.045	**0.761**	0.238
Edges	0.170	0.376	**0.866**
Nodes	0.169	0.384	**0.864**
η_1	0.222	0.341	**0.647**
Eigenvalues	4.715	3.470	2.615

Table 1. Complexity Domain Structure for Avionics Software

Module	Domain1	Domain2	Domain3	ρ	DR Count
1	-0.783	-0.016	0.369	43.36	0
2	-0.782	-0.017	0.368	43.36	0
3	-0.782	-0.017	0.368	43.36	0
4	-0.778	-0.021	0.361	43.37	0
5	-0.777	-0.022	0.359	43.37	0
6	-0.772	-0.027	0.349	43.39	0
7	-0.769	-0.031	0.343	43.40	0
8	-0.763	-0.001	0.314	43.53	0
9	0.069	-0.119	-0.183	49.8	0
10	-0.230	0.456	0.049	49.89	0
11	0.220	-0.078	-1.329	49.92	2
12	0.486	-0.888	-0.526	49.98	9
13	-0.258	0.436	0.412	50.03	1
14	-0.092	0.173	0.125	50.05	0
15	0.259	-0.054	-1.568	50.05	0
16	0.230	0.027	-1.566	50.13	1
17	3.161	3.277	2.557	95.44	9
18	7.575	-5.399	1.667	97.8	25
19	3.752	3.198	1.314	98.80	4
20	3.452	4.460	3.065	103.6	6
21	4.828	2.456	0.269	104.02	4
22	5.989	3.087	6.093	124.72	10
23	7.956	4.452	-4.040	134.83	3
24	8.242	5.139	-0.865	144.42	15

Table 2. Sample Program Modules with Domain Metrics, Relative Complexity and Faults