GP-based software quality prediction

Matthew Evett
Dept. Computer Science & Engineering
Florida Atlantic University
Boca Raton, Florida 33431
(561)297-3459; matt@cse.fau.edu

Taghi Khoshgoftar
Dept. Computer Science & Engineering
Florida Atlantic University
Boca Raton, Florida 33431
(561)297-3994; taghi@cse.fau.edu

Pei-der Chien
Dept. Computer Science & Engineering
Florida Atlantic University
Boca Raton, Florida 33431
chienp@cse.fau.edu

Edward Allen
Dept. Computer Science & Engineering
Florida Atlantic University
Boca Raton, Florida 33431
allene@cse.fau.edu

ABSTRACT

Software development managers use software quality prediction methods to determine to which modules expensive reliability techniques should be applied. In this paper we describe a genetic programming (GP) based system for targetting software modules for reliability enhancement. The paper describes the GP system, and provides a case study using software quality data from two actual industrial projects. The system is shown to be robust enough for use in industrial domains.

1 Introduction

Highly reliable software is becoming an essential ingredient in many systems. Public safety and the fabric of modern life depend on software-intensive systems. We can ill afford for important systems to fail due to inadequate software reliability.

Correcting software faults late in the development life cycle (i.e., after deployment into the field) is often very expensive. Consequently, software developers apply various techniques to discover faults early in development(Hudepohl *et al.* 1996). These reliability improvement techniques include more rigorous design and code reviews, automatic test case generation to support more extensive testing, and strategic assignment of key personnel. While these techniques do not guarantee that all faults are discovered, they greatly decrease the probability of a fault going undiscovered before release. When a fault is discovered, it can be eliminated, and the repaired module possibly resubmitted for further reliability review.

Unfortunately, reliability enhancement can be quite expensive, so these techniques usually cannot be applied to all the software modules comprising a project. Software development managers must attempt to apply reliability improvement techniques only where they seem likely to pay off, that is, to those software modules that appear likely to suffer the most problems. In this paper, we describe a genetic programming (GP) based system for targeting software modules for reliability enhancement.

One of the strongest criticisms of current GP research is that too much of it focuses on toy domains, that GP is not used on real-world problems. In this paper, we present industrial case studies to illustrate our methodology. We apply our GP-based system to software quality data from actual software development projects in two different industrial domains. The results demonstrate that our GP-based system does an excellent job of software quality prediction, and would be a useful tool for managers of large software projects.

Although genetic algorithms have been applied to software testing and software quality modeling for several years, this study is the first application of GP to software engineering that we know of (an extensive survey of on-line evolutionary computation and computer science bibliographies did not reveal any similar studies). Because this study only introduces the application of GP to software engineering, we see many opportunities for future research.

2 Software Quality Modeling

Our previous software quality modeling research has focused on classification models to identify *fault-prone* and *not fault-prone* modules(Khoshgoftaar *et al.* 1996*a*, Khoshgoftaar *et al.* 1996*b*). A software development manager could use such models to target those software modules that were classified as fault-prone for reliability improvement techniques.

However, such models require that *fault-prone* be defined before modeling, usually via a threshold on the number of faults expected, and software development managers often do not know an appropriate threshold at the time of modeling. If the threshold is set too high, no modules may be classified as

fault-prone, even though in any large project some modules are bound benefit from reliability improvement. If, on the other hand, the threshold is set too low, more modules may be classified as fault-prone than resource limitations (manpower, deadlines) will permit for reliability improvement.

In such cases, a prediction of the rank-order of modules, from the least to the most fault-prone, is more useful(Ohlsson and Alberg 1996). With a predicted rank-order in hand, the manager can select, for reliability enhancement, as many modules from the top of the list as resources allow.

Our goal is to develop models that predict the relative quality of each module, characterized as the module's relative ranking among other modules in terms of the number of faults it is likely to produce. We used GP to create models that predict the number of faults expected in each module, but we use these predictions only to rank the modules. Our evaluation of the quality of the generated models is based on ordinal criteria, rather than the amount of error in the predicted number of faults.

Most quality factors, including faultiness, are directly measurable only after software has been deployed. Fortunately, prior research has shown that software product and process metrics(Fenton and Pfleeger 1997) collected early in the software development life cycle can be the basis for reliability predictions. These metrics are quantities such as the number of lines of code, the degree of reuse, the number of faults in previous releases, etc. See Section 3.1 for a detailed list of the metrics used in this paper's case studies. Our GP system uses these metrics as the basis for the models it generates.

Our case studies of the models generated by our GP system are based on actual industrial software development projects. Our case study data consisted of the software metrics for each module in these projects, as well as the number of faults detected in the modules after deployment. The exact methodology used by our GP system to create models on the basis of this data, and our evaluation methodology is explained below.

2.1 Methodology for Evaluating the GP System

An *observation* is a software module represented by a tuple of software measurements, $\mathbf{x}_j$. The dependent variable of a model is the number of faults, $F(j)$, for each observation j. The s-expressions resulting from our GP system are the *models*. Let $\widehat{F}_i(\mathbf{x}_j)$ be the estimate of $F(j)$ by model i. We develop software quality models based on data from a completed past project where measurements and the number of faults are available for each module, using this methodology:

1. Impartially split the available data into *training* and *validation* data sets.
2. Run the GP system multiple times. For each run:
   (a) The GP system has access only to the *training* data. The best-of-run is the model returned as the result of the run. The details of the GP process are described in Section 2.3.
   (b) Use each best-of-run model to predict the number of faults in the *validation* modules, and order them accordingly.
   (c) Evaluate each best-of-run model using ordinal criteria (based on the dependent variable, the actual number of faults observed after deployment,) detailed in Section 2.2.
3. Summarize the model evaluations over the runs for each past project.

Part of our model evaluation (Step 2c) includes a comparison of the ordering obtained by each model to a random ordering, and to the ordering obtained by using the actual observed faults. The first comparison indicates whether a GP result is really different from a random ordering result. (A random ordering emulates a development strategy that randomly selects modules for reliability enhancement treatment, a not uncommon strategy in some development environments. The second comparison (i.e., to the actual ordering) indicates how near the model is to a perfect model.

2.2 Ordinal Evaluation

Each individual of our GP populations (including the best-of-runs) is a model that predicts the number of faults for any software module, given a set of software product measurements for that module. We do not expect a model's prediction of the number of faults of each module to be perfect. In Step 2b of our modeling methodology, we evaluate a model's usefulness by its ability to approximately order modules from the most fault-prone to the least fault-prone.

According to Pareto's Law applied to software engineering, 20% of the modules will typically account for about 80% of the faults. These proportions were true for our case study, where more than 70% of the modules had zero or only one fault. The purpose of the model should be to identify the top 20% of the fault-prone modules. Moreover, resources may be limited for reliability enhancement treatments. Thus, in this study, a manager will probably be interested in reviewing less than 25% of the modules. In this context, let $\mathcal{C}$ be management's preferred set of cutoff percentiles of modules ranked by faults, and let n_c be the number of percentiles in $\mathcal{C}$. In these terms, Pareto's Law implies that the modules above the 80^{th} percentile (i.e., the top 20% of the modules) have 80% of the faults. In the case study, we chose 90, 85, 80, and 75 percentiles. Another project might choose different percentiles, but this set illustrates our methodology.

Let G_{tot} be the total number of actual faults in the validation data set's software modules. Our ordinal evaluation procedure (used in Step 2c for each model) is: Given an individual, i, and a validation data set indexed by j:

1. Determine the perfect ranking of modules, $\mathbf{R}$, by ordering modules according to $F(j)$. Let $R(j)$ be the percentile rank of observation j.
2. Construct a random ranking of the modules, $\mathbf{R}'$. Let $R'(j)$ be the percentile rank of observation j.

3. Determine the predicted ranking, $\widehat{\mathbf{R}}_i$, by ordering modules according to $\widehat{F}_i(\mathbf{x}_j)$. Let $\widehat{R}_i(j)$ be the percentile rank of observation j.

4. For each cutoff percentile value of interest, $c \in \mathcal{C}$:

   (a) Calculate the sum of actual faults, G_c, in modules above the cutoff for $\mathbf{R}$.

   $$G_c = \sum_{j:R(j)\geq c} F(j) \tag{1}$$

   (b) Calculate the sum of actual faults, G'_c, in modules above the cutoff for $\mathbf{R}'$.

   $$G'_c = \sum_{j:R'(j)\geq c} F(j) \tag{2}$$

   (c) Calculate the sum of actual faults in modules above the cutoff, $\widehat{G}_c(i)$, for $\widehat{\mathbf{R}}_i$.

   $$\widehat{G}_c(i) = \sum_{j:\widehat{R}_i(j)\geq c} F(j) \tag{3}$$

5. Calculate the percentage of total faults accounted for by each ranking, namely, G_c/G_{tot}, G'_c/G_{tot}, and $\widehat{G}_c(i)/G_{tot}$. Note that the first of these ratios provides insight into the importance of each percentile level. Comparisons of the other ratios to the first indicate how accurately the other rankings reflect this importance.

6. Calculate how closely the faults accounted for by the random and model rankings match those of the perfect ranking as ratios, G'_c/G_c and $\widehat{G}_c(i)/G_c$. Let

   $$\phi_c(i) = \frac{\widehat{G}_c(i)}{G_c} \tag{4}$$

   The percentage of the actual faults, $\phi_c(i)$, for a percentile level c, is our primary measure of the accuracy of each ranking.

2.3 Details of GP System

Function and terminal sets The function set consists of

$$\mathcal{F} \quad = \quad \{+, -, \times, /, \sin, \cos, \exp', \log\} \tag{5}$$

where $\exp'(x) = \exp(\sqrt{x})$, to lessen the risk of arithmetic overflow. Divide ($/$), exponentiation ($\exp'$), and natural logarithm ($\log$) are modified to protected against invalid inputs. The terminal set, $\mathcal{T}$, consists of the available software product metric variables (the independent variables of the data sets) and the ephemeral random constant generator function, $\Re$, taken from Koza(Koza 1992).

Initial population. We use the *ramped half-and-half* method (50% of individuals are created as full trees, and 50% are created using the *grow* method). The *population size* is constant at $M = 2,000$ individuals. The maximum depth of initial s-expression trees was 6, while the minimum was 3. (The depth limit for subsequent generations was 17 levels.)

Fitness function. In some GP systems, the fitness of an individual can be influenced by that of its ancestors, but here it is not. Thus, we omit the generation of an individual from our notation, below.

Raw fitness of individual i is defined as a logarithmic/exponential function of the absolute errors in predictions of faults.

$$f_{raw}(i) = \log\left(\sum_{j\in\mathcal{O}} \exp\left|(\widehat{F}_i(\mathbf{x}_j) - F(j))\right|\right) \tag{6}$$

where j is an index over all observations, $\mathcal{O}$, in the training data set. This functional form is a starting point for further research. Because our goal is to minimize error, we let *standardized fitness* simply be equal to raw fitness

$$f_{std}(i) = f_{raw}(i) \tag{7}$$

We use adjusted fitness, f_{adj}, as the fitness function in our GP system, using the usual definition:

$$f_{adj}(i) = \frac{1}{1 + f_{std}(i)} \tag{8}$$

Run termination criterion. A run is terminated if an individual i with $f_{adj}(i) = 1$ is encountered, a "perfect" solution to the problem, or when the maximum number of generations is reached. The maximum number of generations is 50.

Selection and population redefinition. The probability of cross-over is $p_c = 0.90$. The probability of reproduction is $p_r = 0.09$. The probability of mutation is $p_m = 0.01$. (All typical values for GP systems (Koza 1992).)

Reproduction. The system uses a straight *fitness-proportionate* method to select candidates for reproduction (i.e., no elitism).

Cross-over. The *tournament* selection method is used to choose each parent for cross-over. The tournament size is $n_t = 7$ individuals(Koza 1994). One cross-over node in each parent's s-expression is independently chosen using a uniform probability distribution. If cross-over creates an offspring that violates the maximum depth of tree parameter, the result is discarded and new cross-over points are chosen until a pair of valid offspring are produced.

Mutation. A straight *fitness-proportionate* method is used to select candidates for mutation. A mutation point is randomly chosen. When mutation produces a tree that violates the maximum depth of tree parameter, the result is discarded and the mutation process is repeated with the same parent until a valid offspring is produced.

Table 1 The Tableau used in the case studies.

Terminal set:	Software product metrics available from each data set and $\Re$, varying over the range [0,1].
Function set:	$\{+, -, \times, /, \sin, \cos, \exp', \log\}$
Initialization:	Ramped half-and-half.
Fitness cases:	188 (CCCS) and 117 (LTS) module observations, each a tuple of numeric software metrics.
Raw fitness:	Log of sum of errors in the predicted number of faults (see 6).
Std. fitness:	same as raw fitness.
Wrapper:	Ranks fitness cases on basis of predicted number of faults.
Parameters:	$M = 2000, G = 50$.
Success Pred.:	Ranking of modules obtained by best-of-generation exactly equals that based on actual faults.
ADFs?:	No

Result designation. We used a technique called "canary functions" to limit the effect of overfitting. We detail canary functions and their efficacy elsewhere(Evett *et al.* 1998).

The canary function should be distinct from the fitness function, but also related toward meeting the same overall goal as the fitness function—in this case, generating a ranking of the modules that maximizes the ordinal evaluation criteria (Section 2.2). For the case studies in this paper, our canary function, C, was defined on an individual i as:

$$C(i) = \operatorname*{avg}_{c \in \mathcal{C}} \phi_c(i) \tag{9}$$

where ϕ_c is as defined in 4, but evaluated over the training data set, and $\mathcal{C}$ is the set of cutoff percentiles of interest. The averaging over the cutoff percentiles provides equal emphasis to each cutoff. The hope is that because the canary function differs from the fitness function, its value will begin to degrade significantly from the fitness function at about the time overfitting occurs. Our experiments (Evett et al. 1998) have supported this hypothesis.

At the end of each generation, the canary function is evaluated on the best-of-generation individual. The result of each run is the best-of-generation individual that obtained the highest value for the canary function. In other words, the result of each run is the best-of-generation individual with the best percentage of actual faults, averaged over the percentile levels of interest. We call this the result individual, i_r. The tableau for the GP system is given in Table 1.

3 Case Study

Our case study consisted of evaluating the GP system on data sets from two industrial software projects.

Table 2 Software Product Metrics

Symbol	Description
η_1	Number of unique operators
N_1	Total number of operators
η_2	Number of unique operands
N_2	Total number of operands
$V(G)$	McCabe's cyclomatic complexity
$V_2(G)$	Extended cyclomatic complexity
	$V_2(G) = V(G) +$ number of logical operators
LOC	Lines of code
$ELOC$	Executable lines of code

3.1 Command, Control and Communications System

The Command, Control and Communications System, CCCS, is a large military data communications system written in Ada. Generally, each module is an Ada package, consisting of one or more procedures. The developers had collected software product metrics from the source code of each module. The value of these metrics comprised the independent variable components of each observation tuple. Table 2 lists the software metrics used in the CCCS case study.

Note that the metrics collected for CCCS do not constitute a canonical set of software metrics—there is no such thing. Software development managers and institutions collect those metrics they feel are important. for their own domains. One of the goals of our system is to provide a methodology that can adapt to a variety of projects, and the product metrics associated with them. This study is an extension of our earlier work (Khoshgoftaar et al. 1998).

We randomly selected 282 modules for our experiment. Applying data splitting, we impartially partitioned this data into two subsets, two thirds of the modules (188) for training the GP system (the training data), and the remaining third (94 modules) for validating the predictive accuracy of the best model from each run (the test data). The top 20% of the modules contained 82.2% of the faults.

3.2 Empirical Results, CCCS

We completed 30 runs of the GP system, using all the observations in the CCCS training data set as the fitness cases. The Lil-gp system, by Zongker et al, formed the kernel of our GP system. The set of software quality models consisted of the result individual, i_r, from each run.

We used each software quality model, i, to predict the number of faults, $\widehat{F}_i(\mathbf{x}_j)$, for each module, j, in the validation data set, and then we ordered the modules by these predicted numbers, forming a ranking, $\widehat{\mathbf{R}}_i$.

We evaluated the rankings at cutoff percentile values of 75%, 80%, 85%, and 90%. If a result individual, i, did not obtain a minimally satisfactory percentage of actual faults, $C(i) \geq C_0$, over the training data set, then it was not included in the analysis below. (Recall that C is based on ϕ_c.)

Table 3 Statistical Summary for CCCS:

Rank $\geq$ %-ile	$\phi_c(i_r)$		
c	Median	Mean	Std Dev
90	82.89%	79.42%	8.18%
85	85.08%	84.82%	9.65%
80	86.87%	84.75%	9.27%
75	87.32%	85.74%	5.72%

Table 4 CCCS.

Rank $\geq$ %-ile	Ranking		
c	Actual, $\mathbf{R}$	Random, $\mathbf{R}'$	Model, $\widehat{\mathbf{R}}_i$
	Faults, $\overline{G}_*$		
	G_c	G'_c	$\overline{\widehat{G}}_c$
90	152	23	120.6
85	181	37	153.52
80	198	47	167.81
75	213	60	182.63
	% of Faults, $\overline{G}_*/G_{tot}$		
90	63.07%	9.65%	50.07%
85	75.10%	15.43%	63.70%
80	82.16%	19.63%	69.63%
75	88.38%	25.01%	75.78%
	% of Actual, $\overline{G}_*/G_c$		
90	100%	15.30%	79.42%
85	100%	20.54%	84.82%
80	100%	23.89%	84.75%
75	100%	28.29%	85.74%

The threshold was $C_0 = 0.70$, which is somewhat less than the accuracy implied by Pareto's Law. The use of the threshold weeded out runs that somehow got stuck in poor local minima in the search space. We made 30 runs of the GP system, of which 5 were unsatisfactory over the training data set. The evaluation below is based on the 25 satisfactory models applied to the validation data set.

Table 3 is a statistical summary over the 25 satisfactory models of the proportion of actual faults $\phi_c(i)$ at each percentile level, c, calculated over the validation data set.

Table 4 shows more detailed evaluation results averaged over the 25 models. (For the table, $G_{tot} = 241$.) For comparison, the table includes evaluation results derived from random models and a perfect model (corresponding to ranking $\mathbf{R}$.) For the random model, we averaged the results of 25 random rankings of the observations in the validation data set to form the ranking, $\mathbf{R}'$.

The first section of Table 4 shows the average sum of faults, "$\overline{G}_*$, for each range of percentiles of fault-prone modules. For example, the top 10% of the modules (ranked by actual faults) account for 152 of the total number of faults in all modules, G_{tot}. Whereas the top 10% of random ranking accounts for only 23 of the faults, on average, the satisfactory models accounted for 120.6 faults. The second section of the table shows the same data as in the first section, but as a proportion of $\overline{G}_*/G_{tot}$. In each of the percentile ranges studied, the models yield rankings much closer to the actual proportion of faults than a random selection ordering. Thus, we conclude that the models are much better than a random sampling strategy. The third section of the table again shows the same data as the first, but as a proportion, $\overline{G}_*/G_c$, of the number of actual faults within each percentile (based on the actual ranking). This is the "average percent of actuals", also known as $\phi_c(i)$, calculated for each ranking mode. For example, on average, the top 10% of the modules in a random ranking account for only 15.30% (i.e. $23/52$) as many faults as are in the top 10% when ranked by the actual number of faults. On the other hand, the average GP-derived model accounts for 79.42% of the actual number. The average percent of actuals was almost 85% of a perfect model (the leftmost column) for the 75, 80, and 85 percentiles, and almost 80% for the 90 percentile. This indicated how closely the averaged sum of predictions for a range of percentiles approaches perfection. The results indicate that this system would be a useful tool for software quality management.

3.3 Legacy Telecommunication System

The second element of our case study was a large legacy telecommunications system, LTS, written by professional programmers in a large organization. This embedded computer application included numerous finite state machines and interfaces to other kinds of equipment. It was written in a proprietary procedural high level language similar to Pascal. The entire system had over $50,000$ procedures; the portion we studied had over $38,000$ procedures in 171 modules.

The LTS data consisted of 19 product metrics for the modules. Prior research (Khoshgoftaar et al. 1996a) with this data set has shown that many of these metrics have little or no relation to faultiness. We used principal component analysis (a statistical method) to transform the product metrics data into five new domain metrics. The value of the five domain metrics and two process metrics that typically correlate strongly with faultiness (development code churn, DEV_NC and debug code churn, FIX_NC) comprised the independent variable components of each observation tuple.

We used all 171 modules in our experiments. As with the CCCS data set, we applied data splitting, impartially partitioning the observations into two subsets: two thirds of the modules (114) as training data, and the other third (57) as testing. The top 20% of the modules contained 81.7% of the faults.

3.4 Empirical Results, LTS

We completed 30 runs of the GP system, using all the training observations of the LTS data as the fitness cases. We generated the same types of rankings ($\mathbf{R}$,$\mathbf{R}'$ and $\widehat{\mathbf{R}}$ as with the CCCS data using the software quality models resulting from the runs. We used the same criteria to exclude "unsatisfactory models" from the statistical analysis. Of the 30 models, 6 did not achieve the threshold of $C_0 = 0.70$, and so were

Table 5 Statistical Summary: 24 satisfactory LTS runs.

Rank $\geq$ %-ile	$\phi_c(i_r)$		
c	Median	Mean	Std Dev
90	92.56%	86.58%	13.85%
85	94.20%	87.67%	14.22%
80	93.41%	87.66%	12.12%
75	91.72%	87.86%	10.95%

Table 6 Model results averaged over the 24 satisfactory LTS runs. $G_{tot} = 56714$**.**

Rank $\geq$ %-ile	Ranking		
c	Actual, $\mathbf{R}$	Random, $\mathbf{R}'$	Model, $\widehat{\mathbf{R}}_i$
	Faults, $\overline{G}_*$		
	G_c	G'_c	$\overline{\widehat{G}}_c$
90	36254	5988	31389
85	41834	8511	36676
80	46335	11300	40617
75	49003	14038	43054
	% of Faults, $\overline{G}_*/G_{tot}$		
90	63.93%	10.56%	55.35%
85	73.77%	15.01%	64.67%
80	81.70%	19.93%	71.62%
75	86.41%	24.75%	75.92%
	% of Actual, $\overline{G}_*/G_c$		
90	100%	16.52%	86.58%
85	100%	20.35%	87.67%
80	100%	24.39%	87.66%
75	100%	28.65%	87.86%

excluded.

Table 5 is analogous to Table 3, showing the proportion of actual faults $\phi_c(i)$ at each percentile level, c, calculated over the LTS validation data set. While Table 6 is analogous to Table 4 and shows a comparison of the value of the rankings obtained by the models against random and perfect rankings.

The results indicate that this system would be a useful tool for software quality management.

4 Conclusions

This study is the first that we know of to apply genetic programming to software quality modeling. In particular, GP can be used to generate software quality models whose inputs are software metrics collected earlier in development, and whose output is a prediction of the number of faults that will be discovered later in development or during operations.

We established ordinal evaluation criteria, rather than the amount of error, for the models produced by the GP system. This is especially appropriate for targeting reliability enhancement activities to the most fault-prone modules.

We conducted industrial case studies of software from a military communications system and a legacy telecommunication system. The GP system used a conventional terminal set, function set, and fitness function(Koza 1994). The performance of the GP-derived models, on actual industrial data, indicated that this system could be a valid software quality management tool.

Acknowledgements

This work was supported in part by a grant from Nortel. The findings and opinions in this study belong solely to the authors, and are not necessarily those of the sponsor.

References

Evett, M.P., T.M. Khoshgoftaar, P.D. Chien and E.B. Allen (1998). Addressing overfitting in genetic programming with canary functions. Technical Report TR-CSE-98-6. Florida Atlantic Univ.. Boca Raton, FL.

Fenton, Norman E. and Shari Lawrence Pfleeger (1997). Software Metrics: A Rigorous and Practical Approach. 2d ed.. PWS Publishing. London.

Hudepohl, John P., Stephen J. Aud, Taghi M. Khoshgoftaar, Edward B. Allen and Jean Mayrand (1996). EMERALD: Software metrics and models on the desktop. IEEE Software **13**(5), 56–60.

Khoshgoftaar, T. M., E. B. Allen, N. Goel, A. Nandi and J. McMullan (1996a). Detection of software modules with high debug code churn in a very large legacy system. In: Proceedings of the Seventh International Symposium on Software Reliability Engineering. IEEE Computer Society. White Plains, NY. pp. 364–371.

Khoshgoftaar, Taghi M., Edward B. Allen, Kalai S. Kalaichelvan and Nishith Goel (1996b). Early quality prediction: A case study in telecommunications. IEEE Software **13**(1), 65–71.

Khoshgoftaar, Taghi M., Matthew P. Evett, Edward B. Allen and Pei-Der Chien (1998). An application of genetic programming to software quality prediction. In: Computational Intelligence and Software Engineering (Witold Pedrycz and Jim F. Peters, Eds.). World Scientific. Singapore. Forthcoming.

Koza, J. (1992). Genetic programming: on the programming of computers by means of natural selection. MIT Press.

Koza, J. (1994). Genetic programming II: Automatic Discovery of Reusable Subprograms. MIT Press. Cambridge, MA.

Ohlsson, Niclas and Hans Alberg (1996). Predicting fault-prone software modules in telephone switches. IEEE Transactions on Software Engineering **22**(12), 886–894.