Generating Multiple Diverse Software Versions with Genetic Programming

Robert Feldt
Department of Computer Engineering
Chalmers University of Technology
Hörsalsv. 11, S-412 96, SWEDEN
feldt@ce.chalmers.se

Abstract

Software fault tolerance schemes often employ multiple software versions developed to meet the same specification. If the versions fail independently of each other, they can be combined to give high levels of reliability. While design diversity is a means to develop these versions, it has been questioned because it increases development costs and because reliability gains are limited by common-mode failures. We propose the use of genetic programming to generate multiple software versions and postulate that these versions can be forced to differ by varying parameters to the genetic programming algorithm. This might prove a cost-effective approach to obtain forced diversity and make possible controlled experiments with large numbers of diverse development methodologies. This paper qualitatively compares the proposed approach to design diversity and its sources of diversity. An experiment environment to evaluate whether significant diversity can be generated is outlined.

1. Introduction

Design diversity, i.e. several diverse development efforts, has been proposed as a technique for generating redundant versions of the same software. These versions are to be employed in structures such as n-version programming, with n versions independently calculating an answer and a voter choosing between them, to give the resulting system the ability to tolerate software faults. The difference, i.e. diversity, in the programs that is generated by the different design methods used for the different versions is called software diversity. The hope is that the diversity in the programs will make them exhibit different failure behavior; they should not fail for the same input and, if they do, they should not fail in the same manner.

There are two main drawbacks with the approach of design diversity: (1) it is not clear how we can guarantee that the developed programs fail independently of each other and (2) the life cycle cost of the software will likely increase. The original idea of n-version programming (NVP) put forward in [1] opted for the specification of the software to be given to different development teams. The teams should independently develop a solution, and this independence between the teams should manifest itself in independent failure behavior. However, software development personnel have similar education and training and use similar thinking, methods and tools. This will lead to common-mode failures, several versions failing for the same input, and limit the diversity that can be achieved. Experimental research ([2]) has shown that there are systems for which the independence assumption is not valid. The strength of using design diversity has thus been questioned.

In [3], the term *random diversity* was proposed to denote the above scenario; generation of diversity is left to chance and arises from differences in background and capabilities of the personnel in the development teams. In contrast to this, they introduced the notion of *enforced diversity*. By listing the known possible sources of diversity and varying them between the different development teams, the software versions can be forced to differ. In [4], Littlewood and Miller showed that the probability that two versions developed with different methodologies would fail on the same input is determined by the correlation between the methodologies. The correlation is a theoretical measure of diversity defined over all possible programs and all possible inputs. Littlewood and Millers calculations set the goal for studies into achieving software diversity: find methodologies with small or negative correlation.

A problem in using design diversity is that life cycle costs can increase. Obviously, the development cost will increase; we have to develop N versions instead of one. In addition to this, maintenance costs increase. Each change or extension to the specifications of the software must be implemented, and possibly even redesigned, in each of the

1089-6503/98 $10.00 © 1998 IEEE

diverse versions. The actual cost increases have been estimated to be near N-fold [5].

This paper introduces a novel approach for developing multiple diverse software versions to the same specification that addresses both the cost and non-independence problems of design diversity. By varying a number of parameters affecting the development of programs, we can force them to differ. The proposed approach uses genetic programming (GP) which, according to [6], is a technique for searching spaces of computer programs for individual programs that are highly "fit" in solving (or approximately solving) a problem. GP evolves programs built from specified atomic parts and adhering to a basic specified structure. Genetic algorithms model evolutionary processes in nature and are studied under the subject of Evolutionary Computation (see for example [7]).

Section 2 introduces genetic programming and section 3 discusses how it can be used to develop diverse software versions. An experiment environment to evaluate how successful this approach is in generating significant diversity is described in section 4. No experiment results are presented. Finally, we give a summary and indicate future directions for this research.

2. Genetic Programming

Genetic algorithms mimic the evolutionary process in nature to find solutions to problems. Genetic programming is a special form of genetic algorithm in which the solution is expressed as a computer program. It is essentially a search algorithm that has shown to be general and effective for a large number of problems.

In the classical view of natural evolution, a population of individuals competes for resources. The most "fit" individuals survive, i.e. they have a higher probability of having offspring in the next generation. This process is modeled in genetic algorithms in which the individuals are objects expressing a certain, often partial or imperfect, solution to the investigated problem. In each generation, each individual is evaluated as to how good a solution it constitutes. Individuals that are good are chosen for the next generation with a higher probability than low-fit individuals. By combining parts of the chosen individuals into new individuals, the algorithm constructs the population of the next generation. Mutation also plays an important part. At random, some parts of an individual are randomly altered. This is a source of new variations in the population.

While a genetic algorithm generally works on data or data structures tailored to the problem at hand, genetic programming works with individuals that are computer programs. This technique was introduced by Koza in [6] and has recently spurred a large body of research ([8]). Kozas programs are trees that are interpreted in software but a number of other approaches exist. For example, in [9] Nordin evolved machine language programs that control a miniature robot.

A number of GP systems are available. To use one of them to solve a particular problem, we must tailor it to the problem. This involves choosing the basic building blocks (called terminals), such as variables and constants, and functions that are to be components of the programs evolved, expressing what are good and bad characteristics of the programs, choosing values for the control parameters of the system and a condition for when to terminate the evolution of programs [6]. The control parameters prescribe, for example, how many individuals are to be in the population, the probability that a program should be mutated and how the initial population of programs should be created.

The major part of tailoring a GP system to a specific problem is to determine a fitness function that evaluates good and bad characteristics of the programs and to develop an environment in which these characteristics can be evaluated. There is no reason to use GP if it is harder to implement an evaluation environment than it is to implement a program solution. However, GP can be used for problems that we can state but for which no solution is known. The fitness function is often implemented via test cases with known good answers. However, the fitness evaluation process is much more general and constitutes any activity taken to evaluate the performance of a program. For example, in [9], the programs are evaluated in a real robot; the ability of the program to avoid obstacles while keeping moving is evaluated and used as a fitness rank.

2.1. Diversity in genetic programming

The term diversity is used with a special meaning in the Evolutionary Computation (EC) community. If the population contains programs that are different, it is said to be diverse. When there is no diversity left in the population, i.e. all programs look and behave the same, the GP run is said to have *converged* to a solution. This can happen before good solutions to the problem have been found and thus different ways to maintain and enhance the diversity are studied (see for example [10]).

Several different measures of diversity have been proposed in the EC community and are classified in [11] into two different classes: genotypic and phenotypic measures. These classes directly correspond to two of the four characteristics of software diversity listed in [3]. Genotypic diversity is called *structural diversity* by Lyu et al. and measures structural differences between the programs. Phenotypic diversity is called *failure diversity* by Lyu et al. and measures differences in the (failure) behavior of the programs.

The diversity remaining in the population when the GP run is terminated can be used to enhance the effectiveness of GP. In [12], Zhang and Joung proposed that a pool of programs, instead of a single one, should be retained from a GP run. The output for certain input is established by applying all the programs in the pool to the input and taking a vote between them to decide the master output, much like an NVP system.

Our approach is distinct from the approach of Zhang and Joung, since we propose that diversity from several runs of a GP system should be exploited and that a systematic variation of the parameters to the GP algorithm should be used to promote diversity. Our goals are also markedly different from the research on measuring diversity. This is primarily done to assess whether a run of a GP system should be stopped because the population has converged ([11]).

2.2. Parameters to a GP system

In the remainder of this paper, we take a pragmatic view of genetic programming. We consider it as a technique for searching a space of programs and view it as a "black box" with three sets of parameters: parameters defining the program space to be searched (program space parameters), parameters defining details about the search (search parameters) and parameters to the evaluation environment (evaluation parameters).

The program space parameters include parameters defining the terminal and function sets and the structure of the programs. These parameters define a space of all possible programs adhering to the specified structure and applying the specified functions to the specified terminals.

The search parameters affect only the result, i.e. the effectiveness, of the searches in the space of programs defined by the program space parameters. Examples of search parameters are the number of programs in the population and the probability that a program should be mutated.

The evaluation parameters define, for example, the number and nature of test cases to be used in evaluation. The strategy for evaluation is also viewed as a parameter. An example of a strategy would be to let the test cases change to test the programs on difficult input values.

It is worth noting that this black-box view frees us from considering only genetic programming. We can consider other algorithms searching a user-definable program space or other algorithms that generate programs. Possible substitutions for GP could be program induction methods or other machine learning algorithms studied in the area of artificial intelligence. Diversity could be found by varying the algorithm used.

3. Software diversity using genetic programming

The output from a run of a GP system is a population of programs that are solutions to the problem stated in the fitness function implemented in the evaluation environment. The solutions are of differing quality; some programs may solve the problem perfectly, others might not even be near solving a single instance of the problem and in between are programs with differing rates of success. The diversity in this population can be exploited, as was done in [12]. However, the amount of diversity available in the population after a GP run will be limited since populations tends to converge to a solution. One way to overcome this might be to rerun the system with the same parameter settings. GP is a stochastic search process, and two runs with the same parameters can produce different results. Diversity might also be achieved by altering parameter values between different runs of the GP system. If we change the search parameters to a GP system, the search might end in different areas of the search space of programs, thus yielding diverse software versions. Furthermore, if we change the program space to be searched by altering the program space parameters, we will get programs using different functions and terminals and adhering to a different structure. Diversity might also be achieved by changing parameters to the evaluation environment. Thus, we propose that diverse software versions are developed by running, re-running and varying parameters to a genetic programming system tailored to the specification for the version(s).

Table 1 outlines the phases in using the proposed method. We start by developing an environment to evaluate the quality of programs, i.e. how well they adhere to the requirements stated in the specification. Thus, upon entering this phase, we need to have a specification at hand. Next, we need to choose which parameters to vary, which values to vary them between and which combinations of parameter values to run with the GP system. Research is needed to evaluate which parameters most affect the diversity of the resulting programs and how to choose their values. The principle for the choice of values should be to include concepts that are thought to be needed to develop a solution, but careful consideration must be made so that the diversity that can be found is not limited.

Phase	Description
1. Evaluation environment	Design a fitness function from the software specification. Implement the fitness function in an evaluation environment.
2. Parameters to vary	Choose which parameters of the GP system and evaluation environment shall be varied.
3. Parameter values	Choose parameter values to vary between.
4. Parameter combinations	Choose the combinations of parameter values to use in the different runs.
5. Generate programs	Run the GP system for each combination of parameter values.
6. Test programs	Test the program versions that have been generated. Calculate measures of diversity.
7. Choose programs	Choose the combination of programs that give the lowest total failure probability for the software fault tolerance structure to be used.

Table 1. Phases of the proposed method for developing diverse program by varying parameters to a genetic programming system

There are large numbers of parameters to a GP system, and most of them can take multiple values, so the number of combinations of parameter values is vast. We think that a systematic exploration of these different combinations should be tried. Statistical methods for the design and analysis of experiments, as for example fractional factorials as described in [13], will likely be needed to this end.

In the next phase (5), the chosen combinations of parameter values is supplied to the GP system which is run to produce the programs. From each run, the best, several or all of the developed program versions can be kept for later testing. If the program generation is not successful, iteration back to phases 2,3 and 4 may be necessary. Upon leaving phase 5, we have a pool of programs. Running a GP system is an automatic process and does not need any human intervention, so the number of programs developed can be large. If we are to use the programs in a specific software fault tolerance scheme, such as an n-version system, we need to choose which programs in the pool to use. Calculating measures of diversity such as the correlation measures in [4] or the failure diversity measure in [3] might be useful in this task and can be calculated from the test data in phase 6.

In [4], systematic approaches to making design choices when employing design diversity were introduced. If we hypothesize that our choices of parameter values are analogous to these design choices, the findings in [4] might be used to choose among the combinations of parameter values. A particular set of design choices is called a design methodology in [4] and, if we take our analogy even further, our GP approach would enable us to try a large number of design methodologies in the same setting. However, it is unclear whether the use of GP or a common evaluation environment limits the diversity to be explored such that the variations in design methodologies are only minor. Research is needed to evaluate this.

In the following, we list sources of diversity when an approach such as NVP is used and identifies which GP parameters relate to these sources. Thereafter, the cost issue of using the proposed GP approach is briefly discussed. Central to the result of applying the described method is that GP can evolve good solutions in the first place. It is not probable that the versions can be used if they fail on a large number of input cases. This issue is further discussed below.

3.1. Comparison of diversity sources

To qualitatively assess the value of the proposed approach, we would like to compare the sources of design diversity with the parameters we can affect in the GP system and what effect on the generated program they might have. Table 2 shows a taxonomy of sources of design diversity and parameters that correspond to these sources. The taxonomy is not intended to be complete but covers the most important aspects mentioned in the literature (see for example [14] and [3]). The taxonomy has been carried over from the Software Metrics area ([15]). Our motivation for this is that what can be measured can be varied and what can be varied, and applies to software and its development, is a potential source of diversity. In [15], Fenton arrives at this taxonomy by viewing a piece of software as a set of activities (processes) using resources to produce artifacts (products). In table 2, a source with leading number 1 is a process, with leading number 2 is a product and with leading number 3 a resource.

We stress that making a comparison like this is not easy; it is not clear-cut how an approach such as genetic programming can be compared with more traditional software development techniques.

Source of design diversity	GP counterpart
1.1 Specification process	Same source
1.2 Design process	Choice of allowed structure, functions and terminals
1.3 Implementation process	Program representation, type of GP system
1.4 Testing process	Evaluation strategy
2.1 Specification products	Same source
2.2 Algorithms	Program space parameters
2.3 Data structures	Functions, Terminals
2.4 Implementation language	Program representation
2.5 Test data	Test cases
3.1 Personnel / Team	No direct counterpart
3.2 Tools	GP System, Diversity in compiler/linker/loader can be similarly exploited

Table 2. Correspondence between sources of diversity in traditional design diversity approaches (based on [15], [14] and [3]) and our proposed GP approach

Processes. The potential diversity arising from different specification processes and/or types also can be used with the GP approach. The difference is that each specification must be implemented in an evaluation environment. The design and implementation processes have no direct counterpart in GP. With GP, we do not explicitly design the programs; they evolve to meet our specification. However, the task of choosing parameters, their values and combinations to be used in the different runs resembles a high-level design activity. We decide not exactly how the program is to be designed but which major concepts can be used.

The potential diversity from using different implementation processes resembles using different types of GP system with, for example, different program representations. An example would be using function trees to represent the programs in one run and using linear representations in another.

The diversity to be found by different testing schemes has no direct counterpart in GP. However, choosing the number and values for the test cases to use in evaluating the programs relates to testing as well as to test data (point 2.5). For different runs, we might choose to concentrate the test cases in a special region of the input data space. Another parameter that resembles alternating the test process would be to allow the test cases to change dynamically.

Products. We cannot directly specify what algorithms and data structures should be used by the GP programs. If we were to give two development teams different functions and terminals to use in their program, however, it might affect what algorithm they used to solve their problem. If the same reasoning applies to our GP system, we would expect the algorithm used in the developed programs to differ for runs with different functions and terminals. The same argument applies for the parameter that determines the permitted structure of the programs. If we dictate that a development team cannot use any subroutines or cannot use recursion, that team might not implement a certain algorithm, forcing them to consider other solutions. In GP, we can introduce functions and terminals that give access to certain types of data structures, such as indexed memory, lists or stacks.

Some studies have shown that using different implementation language can give rise to diversity ([3]). The counterpart in GP is the representation language. This could be one of the earlier mentioned function trees or machine instructions. Other examples are programs implemented with directed acyclic graphs, functional languages or stack-based microinstructions.

Resources. The representation languages in GP are often only intermediary. After the GP run, this intermediate language can be translated into some target language. This makes it possible to leverage diversity available from using different compilation tools, such as compilers, linkers and loaders. The personnel and team sources of design diversity have no direct counterpart in GP. There are many parameters to be set when using GP that have no direct counterparts in ordinary development methodologies. These should not be viewed as purely new ways of adding diversity sources since it is probable that a variation in many of them will have to be restricted considerably for the GP process to find a satisfactory solution.

Summary. There are a large number of parameters in a GP system, and they correspond to some of the sources of diversity in traditional design diversity approaches. Research is needed to evaluate which of the parameters, if any, can be used to force the development of diverse software versions.

We believe that a change in the program space parameters has the greatest potential for generating diversity since it alters the space of programs that are searched. Furthermore, changing these parameters is not difficult and does not incur a large cost and thus should be the focus of a pilot experiment. Changing the parameters of the evaluation environment also shows potential for diversity. However, the cost of doing so is greater and may involve developing alternative

evaluation environments. Finally, changing the search specific parameters should primarily change the rate of success for the GP system. Thus these parameters must be altered to find suitable solutions and may not be available to use for diversity purposes.

3.2. Cost of using genetic programming

Developing one program version in GP is an automatic process. It needs a great deal of processing power but can be speeded up by using parallel computer systems. The evaluation of individuals in a GP population can be done in parallel, and the different runs can be made in parallel when we develop multiple versions. Compared to a traditional approach to design diversity, such as NVP, the cost of development will likely be low; NVP uses human software developers while GP uses processors. This would imply that using GP would decrease the cost of developing an n-version system. The initial cost for the GP approach is higher, however; we may need to try parameter combinations we have not prespecified, and it is unclear how the verification and maintenance costs compare with a traditional approach.

When using GP, we design and implement an evaluation environment from the specification, choose which GP parameters to vary and which values to vary between. With the NVP process, this preparation phase includes administrative tasks such as choosing the design teams, distributing information to them and managing their work. An additional cost in the GP approach is converting the developed versions to a format suitable for execution. The internal representation in the GP system must be converted to binaries for the target machine. However, this cost can be expected to be low since it can be automated.

The cost issue is further complicated if we take verification and maintenance into account. It is unclear how the verification costs of the two approaches compare. The programs developed with GP are generally difficult for humans to read. Their building blocks are the same as in ordinary programs, but there are no comments or design documents and the code can be very complex. In the general case, one cannot expect to debug the programs in the ordinary sense. The program(s) possibly need to be reinserted into the GP system and further developed. Another approach might be to re-run development but emphasizing requirements on the program differently (in the evaluation environment). Similar approaches may be used when maintenance is performed on the n-version system owing to, for example, changing requirements. In this case, the evaluation environment would probably also have to be updated.

3.3. Applicability of genetic programming

We stress that there are serious deficiencies in the theoretical knowledge about genetic programming. The research field is only a couple of years old, and the technique has been applied mostly to toy problems. There is a feeling in the evolutionary computation community that it is time to "step up" and attack real problems, but there is a risk that GP will not scale up to more complex tasks. The applicability of our proposed approach is directly tied to the applicability of GP. If GP can not be scaled up to larger problems, neither can our proposed approach.

At its current level of maturity, GP is probably best suited for controllers, which are small and isolated program components, even though this somewhat contradicts the reason for using software diversity in the first place. The success criteria for control algorithms can be more easily described than, for example, desktop applications since their effects are apparent in the physical world (or in a simulation). Furthermore, GP can be applied even if the underlying control algorithms are poorly understood or not even theoretically known. If we can implement our requirements in an evaluation environment, GP can be applied.

When using the proposed approach, it is crucial that the evaluation environment is free from errors. Since the environment is used to evaluate all programs developed, it is a single point of failure in our development process. This is analogous to the role of the specification in NVP.

4. Pilot experiment environment

We have developed an experiment environment and designed a pilot experiment to investigate whether GP can be used to force the development of diverse software versions. A sketch of the environment is shown in figure 1 below. A pool of programs is developed by a GP system, and each program is tested on the same test set. The failure behavior of the programs can be compared to assess whether any significant diversity has been achieved.

The application problem is to develop a control program for an airplane arrestment system. These systems are used at airfields to bring an airplane to a safe stop in the case of runway overrun. A cable is attached to the incoming airplane and large hydraulic discs are activated to brake the plane. A computer employing a control program sets the break pressure. This application was chosen mainly for simplicity; it is well known at our department, and a simulator is available.

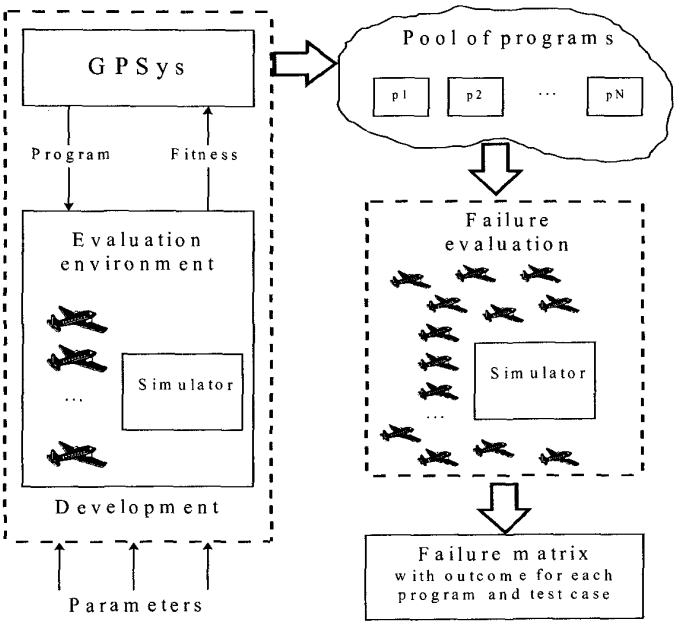

Figure 1. Experiment environment for developing and evaluating airplane arrestment controllers.

The GP development system is built on the GPSys system developed in Java by Adil Qureshi at the University College in London. Simulations are done to evaluate the fitness of the programs arrestments of airplanes with different mass and velocity. The performance of the programs is evaluated according to several criteria extracted from the specification; examples are that the plane does not exceed strength limits or retard the pilot too heavily. Real-valued penalty scores are assigned for each criterion, and these penalties are aggregated to a fitness score. The failure evaluation that takes place after development uses the same simulator to evaluate the programs but, instead of giving real-valued penalty scores, the outcome is simply deemed a failure or a success.

The custom developed parts of the experiment environment are made up of 1125 lines of Java code. They have been tested, and we are now in a position to start the experiments. What remains to be implemented is a measure of diversity between programs and methodologies that can be calculated from the failure evaluation data. We will probably calculate correlations pairwise between methodologies, as described in [4], and/or use the failure diversity measure described by [3]. However, to obtain statistically significant results, we need an analysis of how many programs to develop with each methodology and how many test cases to have in the test set. We are currently working on these issues together with statisticians at our university.

5. Summary and future directions

We have introduced the idea of using genetic programming as a means to force the development of diverse software versions. Genetic programming is a stochastic search technique for searching in spaces of programs. It has a large number of parameters that determine the basic structure, operators and building blocks used in the program versions developed and governs how the programs develop in these basic forms. In addition parameters are available in the environment that is used to evaluate to what extent a program version fulfills criteria stated by the program specification. Choosing values for these parameters parallels, in the terminology of [4], making design choices and results in different development methodologies. The promise of

the proposed approach is that the design choices can be made in a controlled manner, using for example factorial designs, which will allow a search for diverse programs. In addition, the cost of developing multiple versions shows a potential to decrease since multiple versions can be developed once a GP system has been set up. The effect on the life cycle costs of a multi-version system is not known.

Having large numbers of software versions that adhere to the same specification may prove an important step in understanding software diversity and its limitations. The technique can be used on a set of problems not commonly considered by the safety critical computing community: those for which we can state the characteristics of success and failure but for which we know no solution. Furthermore, the approach described in this paper is not limited to genetic programming. It can be used with other techniques for program generation or induction to obtain more sources of diversity.

It is unclear whether the design choices we can affect using genetic programming result in any significant failure diversity in the generated programs. The theory and application of genetic programming is in its infancy and, while much research is ongoing, sufficiently low failure rates might never be obtained. To evaluate whether GP can be used to generate significant diversity, we have developed an experiment environment to conduct a pilot experiment. More research on how to compare two programs or methodologies and assess their diversity under statistical rigor will strengthen this research; these issues are currently being worked on. If these activities are successful, we will go on to conduct larger experiments.

Investigating how new computational models, such as evolutionary computation, affect and can be used in the field of software reliability and fault tolerance is interesting and generates many ideas. We believe that a well of inspiration for building reliable computing systems can be found by studying nature and biological organisms as suggested in [16].

Acknowledgements

The author wishes to acknowledge Jörgen Christmansson, Martin Hiller, Marcus Rimén, Jan Torin and the anonymous referees whose thoughtful remarks increased the quality of this paper. The author strongly opposes the use of the knowledge or ideas in this paper for aggressive military applications.

References

[1] A. Avizienis, and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. *Proc. of COMPSAC-77*, pp. 149-155, 1977.

[2] J. C. Knight, and N. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. on Software Engineering*, 12(1):96-109, January 1986.

[3] M. Lyu, J-H. Chen, and A. Avizienis. Experience in metrics and measurements for N-version programming. *Int. Journal of Reliability, Quality and Safety Engineering*, 1(1):41-62, 1994.

[4] B. Littlewood, and D. R. Miller. Conceptual modelling of coincident failures in multiversion software. *IEEE Trans. on Software Eng.*, 15(12):1596-1614, December 12.

[5] L. Hatton. N-Version design versus one good version. *IEEE Software*, 14(6):71-76, November / December 1997.

[6] J. R. Koza. *Genetic Programming - On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

[7] T. Bäck, U. Hammel, and H-P. Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Trans. on Evolutionary Computation*, 1(1):3-17, April 1997.

[8] J. R. Koza et al. (ed). *Proc. of Second Annual Conf. on Genetic Programming, July 13-16, 1997*. Morgan Kaufmann, San Fransisco, California, 1997.

[9] P. Nordin, and W. Banzhaf. Real Time Evolution of Behavior and a World Model for a Miniature Robot using Genetic Programming. Tech. Report 5/95, Dept. of Computer Science, University of Dortmund, 1995.

[10] C. O. Ryan. *Reducing Premature Convergence in Evolutionary Algorithms*. PhD thesis, Computer Science Department, University College, Cork, July 2, 1996.

[11] W. Banzhaf, P. Nordin, R. Keller and F. Francone. *Genetic Programming - An Introduction*. Morgan Kaufmann, San Fransisco, California, 1998.

[12] B-T. Zhang, and J-G Joung. Enhancing robustness of genetic programming at the species level. *Proc. of Second Annual Conference on Genetic Programming*, July 13-16, 1997, Stanford University, USA, pp. 336-342.

[13] G. E. Box et al. *Statistics for Experimenters - An Introduction to Design, Data Analysis and Model Building*. John Wiley & Sons, New York, 1978.

[14] F. Saglietti. Strategies for the acheivement and assessment of software fault-tolerance. *IFAC 11th World Congress on Automatic Control*, Tallinn, USSR, 1990, pp. 303-308.

[15] N. E. Fenton. *Software Metrics - A Rigorous Approach*. Chapman & Hall, 1991.

[16] A. Avizienis. Building dependable systems: how to keep up with complexity. *Special Issue from FTCS-25 Silver Jubilee*, Pasadena, California, June 27-30, 1995, pp. 4-15.