Generating diverse software versions with genetic programming: an experimental study

R. Feldt

Indexing terms: Design diversity, Fault tolerance, Genetic programming

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. Although 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. The use of genetic programming is proposed to generate multiple software versions by varying parameters of the genetic programming algorithm. An environment is developed to generate programs for a controller in an aircraft arrestment system. Eighty programs have been developed and tested on 10 000 test cases. The experimental data show that failure diversity is achieved, but for the top performing programs its levels are limited.

1 Introduction

One approach to software fault tolerance employs multiple versions of the same software to mask the effect of faults when a minority of versions fails [1]. Design diversity, i.e. several diverse development efforts, has been proposed as a technique for generating these redundant versions. The difference in the programs, which is generated by the different design methods, is called software diversity. The hope is that the diversity in the programs will make them exhibit different failure behaviour: 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 design-diversity approach: it is not obvious if and how we can guarantee that the programs fail independently, and the life-cycle cost of the software will probably increase. The original idea of N-version programming (NVP) opted for the software specification to be given to different development teams [1]. The teams should independently develop a solution, and this independence between the teams should manifest itself in independent failure behaviour. However, software-development personnel have similar education and training and use similar thinking, methods and tools. This can lead to common-mode failures, i.e. several versions failing for the same input, and limit the diversity that can be achieved. Experimental research has shown that there are systems for which the independence assumption is not valid [2]. 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 the 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. The Littlewood and Miller calculations set the goal for studies into achieving software diversity: find methodologies with a 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 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, which addresses both the development cost and non-independence problems of design diversity. 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' for solving (or approximately solving) a problem. GP evolves programs from specified atomic parts that adhere 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]). By varying a number of parameters affecting the development of programs, we can force them to differ.

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

IEE Proceedings online no. 19982444

Paper received 21st July 1998

The author is with the Department of Computer Engineering, Chalmers University of Technology, Hörsalsv. 11, S-412 96, Sweden

essentially a search algorithm that has been shown to be general and effective for a large number of problems.

In the classical view of natural evolution, individuals in a population compete for resources. The most 'fit' individuals survive, i.e. they have a higher probability of having offspring in the next generation. This process is modelled 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 to form 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.

Whereas 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]. Koza 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 carried out 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 the robot 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]). Measuring the diversity in the population is fundamental to this aim.

Several different measures of diversity have been proposed in the EC community and are classified into two different classes: genotypic and phenotypic measures [11]. 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 behaviour of the programs [3].

The phenotypic 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 a certain input is established by applying the programs in the pool to the input and taking a vote between them to decide the master output, similar to an N-version system. Our approach is distinct from the approach of Zhang and Joung, as we propose that diversity from several runs of a GP system should be exploited and that the parameters of the system should be systematically varied to promote diversity. Our goals are also markedly different from the research on measuring diversity in GP populations. The main goal of such research is to decide whether the run should be stopped because the population has converged [11].

2.2 Parameters of a GP system

In the remainder of this paper, we take a pragmatic view of genetic programming. We consider it 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 (PSPs)); parameters defining details about the search (search parameters (SPs)); and parameters of the evaluation environment (evaluation parameters (EPs)).

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

The EPs 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 during evolution 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 with 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 may 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 [12]. However, the amount of diversity available in the population after a GP run will be limited, as populations tend to converge to a solution. One way to overcome this may 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 may also be achieved by altering parameter values between different runs of the GP system. If we change the search parameters of a GP system, the search may end in different areas of the search space of programs. Furthermore, if we change the program space to be searched by altering the PSPs, we will obtain programs using different functions and terminals and adhering to a different structure. Diversity may also be achieved by changing the parameters of the evaluation environment. Thus, we propose that diverse software versions are developed by running, rerunning and varying parameters of a genetic programming system tailored to the specification for the version(s).

Table 1: Phases of proposed procedure for developing diverse programs by varying parameters of a genetic programming system

Phase	Description
1 Evaluation environment	design fitness function from software specification; implement fitness function in an evaluation environment
2 Parameters to vary	choose which parameters of GP system and evaluation environment shall be varied
3 Parameter values	choose parameter values to vary between
4 Parameter combinations	choose combinations of parameter values to use in different runs
5 Generate programs	run GP system for each combination of parameter values
6 Test programs	test program versions that have been generated; calculate measures of diversity
7 Choose programs	choose combination of programs that gives lowest total failure probability for software fault-tolerance structure to be used

3.1 Procedure for developing diverse programs with genetic programming

Table 1 outlines the seven different phases in the procedure we propose. We start by developing an environment to evaluate the quality of programs (phase 1), i.e. how well they adhere to the requirements stated in the specification. Thus, upon entering phase 1, 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. This is done in phases 2, 3 and 4, respectively. Research is needed to evaluate which parameters most affect the diversity. The principle for the choice of values should be to include building blocks, i.e. functions and terminals, that are thought to be needed to develop a solution. Careful consideration must be made so that the diversity is not limited. There are large numbers of parameters of a GP system, most of which can take multiple values, and so the number of combinations of parameter values is vast. We propose that a systematic exploration of these different combinations should be tried. Statistical methods for the design and analysis of experiments, such as, for example, fractional factorials described in [13], are needed to this end.

In the next phase (phase 5), the chosen combinations of parameter values are supplied to the GP system that 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 that does not need any human intervention, and 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 (phase 7). Calculating measures of diversity, such as the correlation measures in [4] or the failure diversity measure in [3], may be useful in this task. The measures 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 hypothesise that our choices of parameter values are analogous to these design choices, the findings in [4] can 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. An experiment to evaluate this is described in Section 4.

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 computers. The evaluation of individuals in a GP population can be made in parallel, and different runs can be made in parallel. Compared with a traditional approach to design diversity, such as NVP, the cost of development will probably be low: NVP uses human software developers, whereas GP uses processors. This implies that using GP would decrease the cost of developing an N-version system. The initial cost for the GP approach may be higher, however, we may need to try parameter combinations that we have not pre-specified, 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 and 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 as 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 and cannot be debugged in the ordinary sense. The programs may need to be reinserted into the GP system and further developed. Another approach may be to re-run development but emphasise requirements of the program differently. Similar approaches can be used when maintenance is performed on the *N*-version system, owing to, for example, changing requirements.

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 few 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 cannot be scaled up to larger problems, neither can our proposed approach.

At its current level of maturity, GP is probably best suited to small and isolated program components, such as simple controllers, 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, as 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 the proposed approach is used, it is crucial that the evaluation environment is free from errors. As 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 Description of experiment

We have used a genetic programming system to develop 80 program variants from the same specification. The programs were developed automatically by a custom-developed system running on a SUN Enterprise 10000 with the Sun Solaris OS and Java Development Kit 1.2. The GP system was run five times for 16 different settings of parameters. The resulting 80 programs were subjected to the same 10 000 test cases, and their failure behaviour was analysed to assess the diversity of the programs. Fig. 1 shows a diagram of the experiment environment. Below we describe the target system, the GP system, the design of the experiment and the testing procedure. A more thorough description is given in [14].

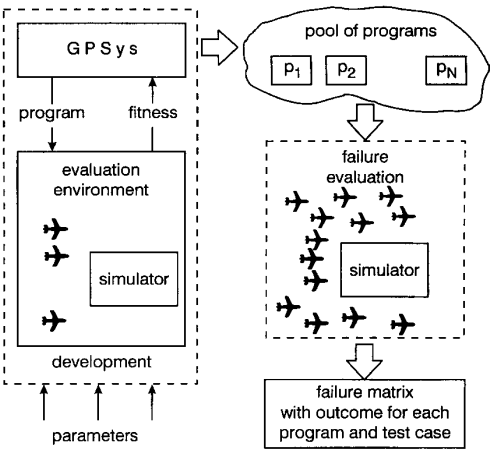

Fig. 1 *Experiment environment for developing and evaluating aircraft arrestment controllers*

4.1 Target system

The target system is designed to arrest aircraft on a runway. Incoming aircraft attach to a cable, and the system applies pressure on two drums of tape attached to the cable. A computer that determines the break pressure to be applied controls the system. By dynamically adapting the pressure to the energy of the incoming aeroplane, the program should make the aircraft come to a smooth stop. The requirements on a system like this can be found in [15]. The system has been used in other research at our department, and a simulator simulating aircraft with different mass and velocity is available. The system is more fully described in [16].

The main function of the system is to brake aircraft smoothly, without exceeding the limits of the braking system, the structural integrity of the aircraft or the pilot in the aircraft. The system should cope with aircraft with maximum energy of 8.81×10^7 J and mass and velocity in the range 4000–25 000 kg and 30–100 ms^{-1}, respectively. More formally, the program should [Note 1] (with name of corresponding failure class given in parentheses)

- stop aircraft at, or as close as possible to, a target distance
- stop the aircraft before the critical length of the tape (335 m) in the system (overrun)
- not impose a force on the cable or tape of more than 360 kN (cable)
- not impose a retarding force on the pilot corresponding to more than 2.8 g (retardation)
- not impose a retarding force exceeding the structural limit of the aircraft, given for a number of different masses and velocities in [15] (hookforce)

The programs are allowed to use floating point numbers in the calculations. They are invoked for each 10 m of cable and calculate the break pressure, for the following 10 m, given the current amount of rolled out cable and angular velocity of the tape drum.

An existing simulator of the system has been imported from C to Java. It implements a simple mechanical model of the aeroplane and braking system and calculates the position, retardation, forces and velocities in the system. It does not model the inertia in

Note 1: Our system adopts the requirements of [15], with the addition of the allowed ranges for mass and velocity and a critical length of 335 m, 950 feet in [15])

the hydraulic system or oscillatory movement of the aircraft due to elasticity in the tape. The simulator has been set to simulate braking with a time step of 62.5ms.

4.2 Genetic programming system

Our development system is built on top of the GPSys genetic programming system, written in Java by Adhil Quereshi at University College, London. The programs in this system are function trees that are interpreted when used in braking the aircraft. During evolution, GPSys invokes the simulator to evaluate the fitness of programs. Values from the simulation are used to assign penalty values to the four fitness criteria. The penalties are assigned in a non-linear fashion, with high values when the program fails on the criteria. For the overrun criterion:

- if the stop position of the aircraft is larger than the critical length of the system, a basic penalty is assigned; the basic penalty was chosen as 80% of the maximum penalty for the criterion.
- a guiding penalty is assigned if the velocity of the aircraft is larger than zero on the critical length; this is to distinguish programs that almost succeeded in braking the aircraft from programs that have not even tried and 'guides' the programs in the direction of good performance; the basic penalty was chosen as 20% of the maximum penalty for the criterion.
- if the aircraft comes to a halt, a linear penalty is assigned; it diminishes from its maximum value at position 0 up to the target distance and then increases up to its maximum again at the critical length; this is to ensure that a halt position close to the target distance will give the program a low penalty; the maximum amount of linear penalty is a parameter of the system but should be much smaller than 80%.

The penalties for the other criteria are assigned in a similar manner. For more details, consult [14]. The penalty values on the four criteria are summed to give the total fitness for the test case. The total fitness of the program is the sum of the fitnesses on all the test cases. A perfect program would obtain a fitness value of zero.

4.3 Testing procedure

After each run of the GP system, the best program is evaluated on 10000 test cases, evenly spread over the range of valid masses and velocities. Dividing the range of allowed mass into 100 locations 212.12kg apart, generates these test cases. For each mass, a maximum velocity is calculated, so that the resulting energy does not exceed the 8.81×10^7 J specified in [15]. The range (30, maximum velocity for this mass) is divided into 100 velocities, and a total of $100 \times 100 = 10000$ test cases result.

4.4 Experimental design

The discussion in [17] argued that the program space defining parameters (PSPs) should have the largest effect on the diversity of the resulting programs. The parameters of the evaluation environment (EPs) should also have an effect, whereas the search parameters (SPs) may primarily affect the effectiveness of the GP system. In accordance with this, we have chosen to vary four PSPs, three EPs and one SP. Many of these parameters can take multiple values, giving rise to an

Table 2: Description of parameters varied in experiment and their levels

Factors	Levels	Type	Description	Anticipated effect/Motivation
A	−1	PSP	no effect.	for comparison of values during braking
	1		the statement If, and operators LE, And and Not can be used in programs	
B	−1	PSP	no effect.	for oscillatory and/or damping behaviour
	1		the functions Sinus and Exp can be used in the programs	
C	−1	PSP	the average velocity, average retardation and index to current checkpoint can be used in programs	for structural diversity; average velocity and retardation are pre-calculated before they can be used in programs
	1		the angular velocity, current time since start of braking, previous angular velocity and time of previous checkpoint can be used in programs	
D	−1	PSP	programs cannot use any subroutines	For greater program complexity without need for one long program
	1		two subroutines (automatically defined functions) can be used in program; they are evolved in same manner as rest of program	
E	−1	EP	maximum penalty on retardation failure criterion is 1000.0	force programs to find solutions that solve retardation criterion with higher priority than other criteria
	1		maximum penalty on retardation failure criterion is 2000.0.	
F	−1	EP	linear penalties are not used	without linear penalties, fitness only expresses 'amount' of failure; performance on non-failure aspects is not measured
	1		linear penalties are used, and maximum penalty of 30.0 is assigned to each failure criterion	
G	−1	EP	25 test cases uniformly spread over range of possible values for mass and velocity are used to evaluate fitness during evolution	uniform spreading of test cases 'samples' all parts of possible input cases; random spreading can give both easier and more difficult test cases
	1		25 test cases chosen randomly for each run of the GP system are used to evaluate fitness during evolution	
H	−1	SP	probability of mutation is 0.05	initial experiments indicated that high values might be beneficial
	1		probability of mutation is 0.6	

enormous number of combinations. To make a study feasible, we have confined the parameters to two levels, represented by –1 and 1. The parameters and their levels are listed in Table 2. All other parameters of the system were held constant during the experiment. Each run used 1000 programs in the population and ran for 200 generations.

The result of a GP run is not deterministic, and we need replicated runs for each setting of the parameters. The number of unique settings of eight two-valued parameters is 256, but we used a $2^{(8-4)}$ fractional factorial of resolution IV to reduce this to 16 [13, 15]. The settings of the parameters are shown in Table 3. Once the order in which to run the 80 experiments had been randomised, the experiment was started. The system ran the 80 runs over the course of five days, without any human intervention.

Table 3: Fractional factorial design of experiment with levels for the parameters at each setting

Setting	A	B	C	D	E = B*C*D	F = A*C*D	G = A*B*C	H = A*B*D
1	–1	–1	–1	–1	–1	–1	–1	–1
2	1	–1	–1	–1	–1	1	1	1
3	–1	1	–1	–1	1	–1	1	1
4	1	1	–1	–1	1	1	–1	–1
5	–1	–1	1	–1	1	1	1	–1
6	1	–1	1	–1	1	–1	–1	1
7	–1	1	1	–1	–1	1	–1	1
8	1	1	1	–1	–1	–1	1	–1
9	–1	–1	–1	1	1	1	–1	1
10	1	–1	–1	1	1	–1	1	–1
11	–1	1	–1	1	–1	1	1	–1
12	1	1	–1	1	–1	–1	–1	1
13	–1	–1	1	1	–1	–1	1	1
14	1	–1	1	1	–1	1	–1	–1
15	–1	1	1	1	1	–1	–1	–1
16	1	1	1	1	1	1	1	1

Table 4: Number of failures for each of 80 versions, average and average success probability for each setting of parameters

Setting	A	B	C	D	E	Average	P_{succ}, %
1	1083	708	813	1327	1475	1081.2	89.19
2	**591**	2100	648	1746	831	1183.2	88.17
3	893	1275	888	1016	1150	1044.4	89.56
4	2205	2694	1644	2639	1240	2084	79.16
5	**588**	670	1657	**559**	1159	926.6	90.73
6	801	**559**	965	753	2968	1209.2	87.91
7	**499**	697	575	1054	985	762	92.38
8	998	**586**	1479	767	713	908.6	90.91
9	3164	2429	3609	2374	2408	2793.2	72.07
10	1200	1433	1212	1063	2112	1404	85.96
11	809	1432	1140	870	1027	1055.6	89.44
12	1726	755	1782	2255	1789	1661.4	83.39
13	811	996	852	754	1578	998.2	90.02
14	**392**	1177	2240	1026	942	1155.4	88.45
15	1108	1053	**630**	2388	**560**	1147.8	88.52
16	2946	1111	1005	827	954	1368.6	86.31

5 Experimental results

For each test case executed, a trace of the braking of the aeroplane is returned from the simulator. Four values are extracted from this trace to classify the behaviour of the program: halt distance of the aircraft, maximum force in the cable, maximum retardation force on the hook, and maximum retardation during the braking. These values correspond to the four fitness criteria above. We record a failure for a particular version on a particular test case if any value exceeds its limits. Failure is indicated by one (1), and success is indicated by zero (0), and these binary values are collected into a failure behaviour vector giving the failure behaviour on a particular test case.

The quality of the 80 programs varies highly. Table 4 shows the observed failure rates of the versions. The average number of failures is 1298.96 (probability of success P_{succ} = 87.01%), with a standard deviation of 712.89 failures. The best program failed on 392 test cases (P_{succ} = 96.08%), and the worst failed on 3609 (P_{succ} = 63.91%). The top ten performing programs are shown in bold face in Table 4. The average number of failures among them is 553.90 (P_{succ} = 94.46%), with a standard deviation of 65.93 failures.

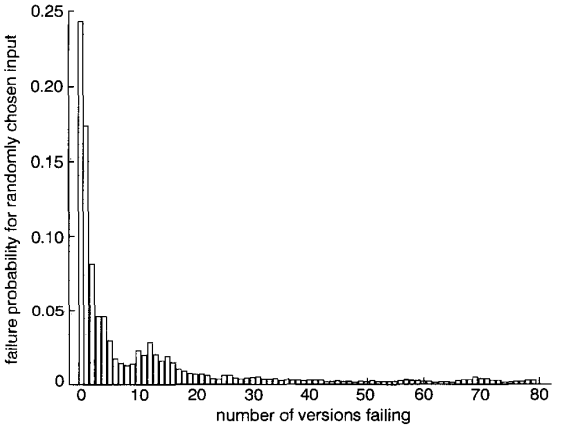

Fig.2 *Probability of failure of n versions for randomly chosen input*

Many programs failed on the same test case. Fig. 2 shows the probability that n of the 80 versions fail on a randomly chosen test case among the 10 000 test cases. There are no test cases for which all programs fail, but many test cases seem to be troublesome for the programs. For example, there are 22 test cases on which 79 of the programs fail, and 24 test cases on which 78 fail. This indicates that some test cases are more difficult than others. The variability in difficulty is shown in a contour plot in Fig. 3. Darker areas show regions where more programs fail.

The structural diversity of the programs varies. A simple measure of this diversity was recorded: the size of the program trees. The average size is 100.2 nodes in the tree, with a standard deviation of 82.87. The maximum size is 459, and the minimum size is 17. No correlation was found between the size of the programs and the number of failures they exhibited (correlation coefficient 0.05). The average size of the top ten programs is 84.8, with a standard deviation of 46.07. The maximum size is 185, and the minimum is 38.

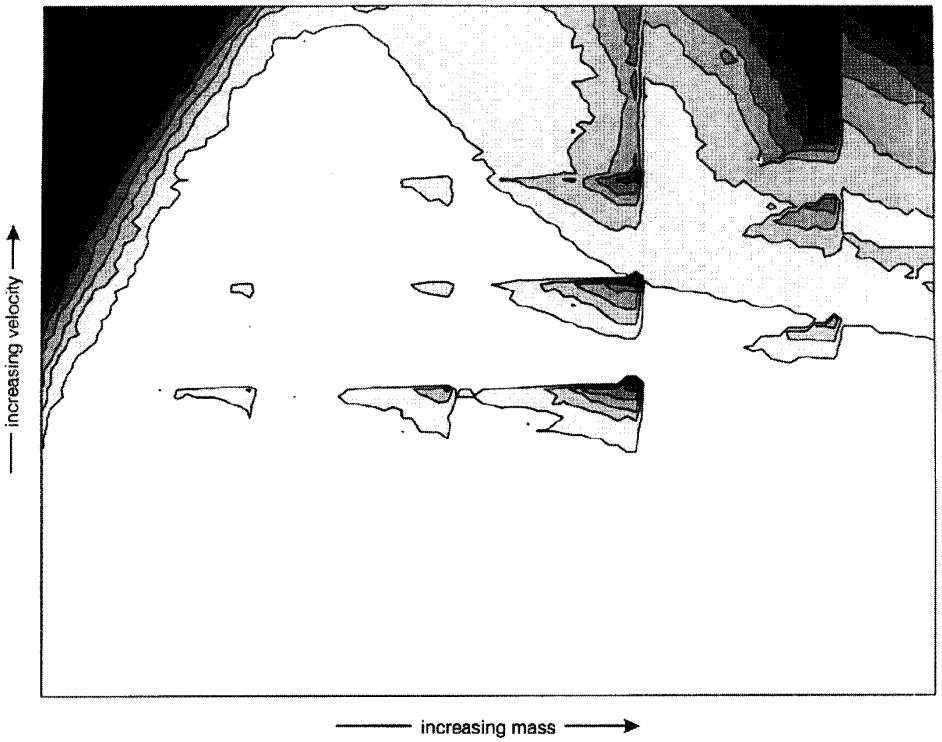

Fig.3 *Test case difficulty*

6 Evaluation of results

In the following, we evaluate the failure diversity, test case difficulty variability and performance of three-version systems constructed from the programs. The failure diversity is evaluated between the individual programs and between the different methods defined by our 16 different settings of parameters. A statistical test is performed to evaluate whether varying the parameters of the system generates diversity.

6.1 Failure diversity

Different measures of diversity have been proposed in the literature. In [4], Littlewood and Miller propose that the amount of diversity between two design methods should be measured using the correlation coefficient of the joint distribution of their failures. Their measure is theoretical, as it should be applied for all input cases and programs that can be developed with the methods. We have used it in the same way that Littlewood and Miller use it in their examples: by disregarding difficult issues of statistical sampling [4]. Another failure diversity measure was used in [3]. It is defined as the number of distinct failures divided by the total number of failures, and below we denote it LFD.

6.1.1 Between programs: The diversity measures were calculated pairwise for all 80 programs. The minimum correlation [Note 2] was –0.2123, and, of the 3160 correlations, 193 (6.11%) were below zero. The maximum LFD was 0.9894. This is encouraging, as low correlations and high failure diversity indicate that taking a vote among versions can mask the effects of failures. However, if we consider only the top ten programs, the picture is different. The lowest pairwise correlation found is 0.5495, and the highest pairwise LFD is 0.5965.

6.1.2 Between methods: We have calculated the 120 inter-method correlations, where each setting of the parameters of the GP system is considered a unique method. The majority of the correlations are high, but 11 are below 0.20, and two are negative. This was surprising and indicates that the variability of difficulty of the test cases can be overcome and the program versions can show better than independent failure behaviour. However, the majority of methods involved in the lowest inter-method correlations are the ones having the highest average failure rate. Thus, even if we pick programs for N-version systems from methods showing low correlation, the failure rate of the system will probably not equal that of the top performing programs.

6.1.3 Inter-method against intra-method diversity: To evaluate whether diversity can be obtained by altering the parameters of the GP system, we wanted to assess whether there is more diversity between versions in different methods than within the same method. To this end, we used the following procedure:

- randomly choose one method and two distinct programs (A_1 and A_2) from it
- randomly choose a program not in A and call it B_1
- compare the diversity between A_1 and A_2 with the diversity between A_1 and B_1. If the latter is larger than the former, the outcome of the test is called positive.

Under the null hypothesis that there is no difference in diversity between the versions due to the different

Note 2: Calculated as the correlation between the failure behaviour vectors; this is a special case of the Littlewood and Miller correlation measure, when there is only one version in each method

methods used, the number of positive outcomes when the above procedure is repeated should be binomially distributed, with n = the number of repetitions of the test and $p = 0.5$.

For each of the two diversity measures, we performed 2400 test procedures. For the correlation measure, 1534 positive outcomes were recorded and, for the failure diversity measure, 1524 positive outcomes were recorded. The null hypothesis could be rejected at the 0.01 level for both of the diversity measures (both with p-value $< 10^{-10}$), and we favour the hypothesis that the failure diversity is larger between versions developed with different settings of the GP parameters than between versions with the same settings.

The top ten programs do not make up a sufficient data record on which to perform this hypothesis testing. Instead, the procedure was applied on the 11 methods with an average failure rate below the total average [Note 3]. In 2400 repetitions of the procedure, 1350 (1310 with the failure diversity measure of Lyu *et al.*) positive outcomes were recorded. Thus, the null hypothesis could still be rejected at the 0.01 level ($p = 4.97 \times 10^{-10}$ and $p = 3.85 \times 10^{-6}$, respectively).

6.2 Test case difficulty variability

Detailed study of the test case difficulty variability shown in Fig. 3 reveals that there are three main areas of difficulty. Visually, these areas are located in the upper left corner, in equidistant clusters in the centre and in the upper right corner, respectively.

For the upper left corner, where aircraft have high velocity and low mass, the programs generally fail on the retardation criteria. It seems plausible to assume that these failures arise because the programs do not properly measure and/or use a notion of the mass of the incoming aircraft in their control algorithm. The failures in the centre area are mainly due to failure on the hookforce criteria. The requirements in [15] stated the maximum allowed hookforce for certain points with specified mass and velocity. The clusters of failing programs seen in the centre of Fig. 3 are located below (lower velocity) and to the left (lower mass) of these points. These are the areas where the energy of the aircraft is at a maximum for the requirement of maximum hookforce. In this light the clusters can almost be expected to appear. The failures in the upper right corner are made up of failures on the hookforce and halt-distance criteria. The former can be explained by the same reasoning as above and the latter arises because the energies of the aircraft take on their largest values this area. If the programs do not exert a high enough brake pressure at the start of braking they will not have time to brake the aircraft before the critical length.

6.3 3-version systems constructed from the programs

We constructed 3-version systems from our programs. The majority vote between the failure behaviours of the programs was taken as the outcome if voting had been applied during the brakings. We believe that this is a worst-case scenario, but have not investigated it further. If voting is applied in the checkpoints during the braking failures that occur at different points in time might be masked. For example, this would happen if program 1 exceeds the maximum allowed retardation early in the braking but after that performs well and program 2 have the opposite behaviour (good performance early, failing in the end). With our post-run voting the behaviour of the system would be deemed a failure regardless of the fact that actual voting at the checkpoints would mask the failures.

We considered all 120 possible N-version systems consisting of three programs taken from the top ten programs. In 41(34.17%) of them the failure rate of the system was lower than the minimum failure rate of the individual programs. The best improvement found, compared to the minimum failure rate of the individual programs in the system, was a decrease from 559 to 444 failures (20.57%).

7 Discussion and conclusions

We have proposed a procedure for developing diverse software versions and have shown that the versions can be forced to be diverse by varying parameters to the genetic programming algorithm used to develop the programs. The low levels of inter-method diversity found between some settings of the parameters were surprising. It indicates that voting in an N-versions system could mask individual program failures. However, the methods giving the lowest correlations are also the ones with the highest failure rates, and the correlation cannot be exploited to give failure rates lower than the top performing programs. The diversity levels found in the top performing programs were much lower. Further analysis will be conducted to find out if the poor performance can be said to cause the high diversity.

The observed behaviour might be explained by the special nature of the target system. It shows a high level of input case difficulty variability, which is known to limit the amount of exploitable diversity [4]. The difficulty arises from the fact that higher energies put more stress on the system. Whether this amount of input case variability is typical is not known. Further experiments with other target systems could shed light on this issue.

In our experiments, we have varied eight parameters of the GP system. It is possible that different choices of parameters and their values would give different results. For example, the apparent problem of the programs to brake light aircraft with high velocities may be overcome by letting them use an indexed memory, making comparisons between values at different checkpoints possible.

Further analysis of the experimental data should focus on revealing the effects of different parameters on the failure rate and diversity of the generated programs. The fractional factorial experimental design we have employed is well suited to this end. The diversity may also be limited by choices we made for the basic system design. For example, better results may be achieved if the programs calculate the break pressure more frequently during braking. This would probably require a smaller time step in the simulation and lead to higher performance demands during program development. We will investigate tools for compiling the experimental environment to native machine code. The increase in performance will allow larger populations and longer runs of the GP system, possibly resulting in better program performance.

Our classification of a failure can be considered worst-case. When comparing the failure behaviour of

Note 3: Hence, methods 4, 9, 10, 12 and 16 were excluded

programs, we do not compare each failure criterion individually. Thus, diversity in the way the programs fail is not accounted for, even though it could be exploited in a system employing fault masking. Our failure classification does not take the time aspect of the program behaviour into account. A situation can easily be envisioned where two programs both exceed the maximum allowed hook force but at different times. This faulty behaviour could be masked by an N-version system. It would be interesting actually to construct N-version systems from our programs and evaluate their failure behaviour.

Further experimentation with the existing system will be conducted, as it mainly amounts to initiating runs and collecting and analysing data; the development of the programs requires no human activity. 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 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 would be interesting to extend our work and compare different techniques of this kind.

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 [18].

8 Acknowledgments

The author wishes to acknowledge the contributions made by Marcus Rimén, Susanne Bolin, Martin Hiller, Jörgen Christmansson, Klas Hjelmgren and Jan Torin, whose thoughtful remarks improved the quality of this paper. The author strongly opposes the use of the knowledge or ideas in this paper for aggressive military applications.

9 References

1 AVIZIENIS, A., and CHEN, L.: 'On the implementation of N-version programming for software fault-tolerance during program execution', Proceedings of COMPSAC-77, 1977, pp. 149–155
2 KNIGHT, J.C., and LEVESON, N.: 'An experimental evaluation of the assumption of independence in multiversion programming', *IEEE Trans. Softw. Eng.*,, **12**, (1), pp. 96–109
3 LYU, M., CHEN, J-H., and AVIZIENIS, A.: 'Experience in metrics and measurements for N-version programming', *Int. J. Reliability, Quality & Safety Eng.*,, **1**, (1), pp. 41–62
4 LITTLEWOOD, B., and MILLER, D.R.: 'Conceptual modelling of coincident failures in multiversion software', *IEEE Trans. Softw. Eng.*,, **15**, (12), pp. 1596–1614
5 HATTON, L.: 'N-version design versus one good version', *IEEE Softw.*,, **14**, (6), pp. 71–76
6 KOZA, J.R.: 'Genetic programming - on the programming of computers by means of natural selection' (MIT Press, Cambridge, Massachusetts, 1992)
7 BÄCK, T., HAMMEL, U., and SCHWEFEL, H-P.: 'Evolutionary computation: comments on the history and current state', *IEEE Trans. Evolut. Comput.*,, **1**, (1), pp. 3–17
8 KOZA, J.R.: Proceedings of second annual conference on *Genetic programming*, 13-16 July 1997, (Morgan Kaufmann, San Fransisco, California)
9 NORDIN, P., and BANZHAF, W.: 'Real time evolution of behaviour and a world model for a miniature robot using genetic programming'. Technical Report 5/95, Department of Computer Science, University of Dortmund, 1995
10 RYAN, C.O.: 'Reducing premature convergence in evolutionary algorithms'. PhD dissertation, Computer Science Department, University College, Cork, 1996
11 BANZHAF, W., NORDIN, P., KELLER, R., and FRANCONCE, F.: 'Genetic programming - an introduction' (Morgan Kaufmann, San Fransisco, California, 1998)
12 ZHANG, B.-T., and JOUNG, J.-G.: 'Enhancing robustness of genetic programming at the species level'. Proceedings of second annual conference on *Genetic programming*, Stanford University, USA, July 1997, pp. 336–342
13 BOX, G.E., HUNTER, W.G., and HUNTER, J.S.: 'Statistics for experimenters - an introduction to design, data analysis and model building' (John Wiley & Sons, New York, 1978)
14 FELDT, R.: 'An experiment on using genetic programming to generate multiple software variants'. Technical Report 98-13, Department of Computer Engineering, Chalmers University of Technology, 1998
15 US Air Force - 99: Military Specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction, MIL-A-38202C, Notice 1, US Department of Defense, 1986
16 CHRISTMANSSON, J.: 'An exploration of models for software faults and errors'. PhD Dissertation, Department of Computer Engineering, Chalmers University of Technology, 1998
17 FELDT, R.: 'Generating multiple diverse software versions using genetic programming'. Euromicro conference 1998, Västerås, Sweden, August 1998, pp. 387–394
18 AVIZIENIS, A.: 'Building dependable systems: how to keep up with complexity'. Special Issue from FTCS-25 Silver Jubilee, Pasadena, California, June 1995, pp. 4–15