Paper 3.

Robert Feldt. *An Experiment on Using Genetic Programming to Develop Multiple Diverse Software Variants*, Technical Report no. 98-13, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, September 1998.

This report includes the two previously published papers:

Robert Feldt. *Generating Multiple Diverse Software Versions with Genetic Programming - an Experimental Study*, IEE Proceedings - Software, vol. 145, issue 6, pp. 228-236, December 1998.

Robert Feldt. *Generating Multiple Diverse Software Versions with Genetic Programming*, Proceedings of the 24th EUROMICRO Conference, Workshop on Dependable Computing Systems, pp. 387-396, Västerås, Sweden, August 1998.

An Experiment on Using Genetic Programming to Generate Multiple Software Variants

Robert Feldt

Technical Report 98-13, Department of Computer Engineering, Chalmers University of Technology, Sweden, October 1998.

Abstract

Software fault tolerance schemes often employ multiple software variants developed to meet the same specification. If the variants fail independently of each other, they can be combined to give high levels of reliability. While design diversity is a means to develop these variants, 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 variants by varying parameters to the genetic programming algorithm. We have developed an environment to generate programs for a controller in an aircraft arrestment system. Eighty programs have been developed and tested on 10000 test cases. The experimental data shows 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 variants of the same software to mask the effect of faults when a minority of variants fails [Avizienis77]. Design diversity, *i.e.*, several diverse development efforts, has been proposed as a technique for generating these redundant variants. 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 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 obvious if and how we can guarantee that the programs fail independently and (2) the life cycle cost of the software will likely increase. The original idea of N-variant programming (NVP) opted for the specification of the software to be given to different development teams [Avizienis77]. 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 may lead to common-mode failures, *i.e.*, several variants 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 [Knight86]. The strength of using design diversity has thus been questioned.

In [Lyu94], 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 variants can be forced to differ. In [Littlewood89], Littlewood and Miller showed that the probability that two variants 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 variants 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 variants. The actual cost increases have been estimated to be near N-fold [Hatton97].

This paper introduces a novel approach for developing multiple diverse software variants to the same specification that addresses both the development cost and non-independence problems of design diversity. The proposed approach uses genetic programming (GP) which, according to [Koza92], 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 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 [Bäck97]). By varying a number of parameters affecting the development of programs, we can force them to differ.

Section 2 introduces genetic programming and section 3 discusses how it can be used to develop diverse software variants. The experiment that have been performed is described in section 4 followed by the experimental results in section 5. Section 6 evaluates the results. Finally, we conclude and indicate future work.

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, 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 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 [Koza92] and has recently spurred a large body of research [Koza97]. Kozas programs are trees that are interpreted in software but a number of

other approaches exist. For example, in [Nordin95] 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 [Koza92]. 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 [Nordin95], 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 [Ryan96]). 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 [Banzhaf98]. These classes directly correspond to two of the four characteristics of software diversity listed in [Lyu94]. 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 phenotypic diversity remaining in the population when the GP run is terminated can be used to enhance the effectiveness of GP. In [Zhang97], 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-variant 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 the parameters to 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 [Banzhaf98].

2.2 Parameters to 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, PSP), parameters defining details about the search (search parameters, SP) and parameters to the evaluation environment (evaluation parameters, EP).

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.*, 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 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 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 [Zhang97]. However, the amount of diversity available in the population after a GP run will be limited since populations tend 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. 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 variants are developed by running, re-running and varying parameters to a genetic programming system tailored to the specification for the variant(s).

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 I), *i.e.*, how well they adhere to the requirements stated in the specification. Thus, upon entering phase I, 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 II, III and IV, respectively. Research is needed to evaluate which parameters most affect the diversity. The principle for choice of values should be to include building blocks, *i.e.*, functions and terminals, which 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 to a GP system, and most of them can take multiple values, 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 [Box78], is needed to this end.

Phase	Description
I. Evaluation environment	Design a fitness function from the software specification. Implement the fitness function in an evaluation environment.
II. Parameters to vary	Choose which parameters of the GP system and evaluation environment shall be varied.
III. Parameter values	Choose parameter values to vary between.
IV. Parameter combinations	Choose the combinations of parameter values to use in the different runs.
V. Generate programs	Run the GP system for each combination of parameter values.
VI. Test programs	Test the program variants that have been generated. Calculate measures of diversity.
VII. 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 procedure for developing diverse programs by varying parameters to a genetic programming system

In the next phase (phase V), the chosen combinations of parameter values are supplied to the GP system which is run to produce the programs. From each run, the best, several or all of the developed program variants can be kept for later testing. If the program generation is not successful, iteration back to phases II, III and IV may be necessary. Upon leaving phase V, 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-variant system, we need to choose which programs in the pool to use (phase VII). Calculating

measures of diversity such as the correlation measures in [Littlewood89] or the failure diversity measure in [Lyu94] might be useful in this task. The measures can be calculated from the test data generated in phase VI.

In [Littlewood89], 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 [Littlewood89] might be used to choose among the combinations of parameter values. A particular set of design choices is called a design methodology in [Littlewood89] 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 below.

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 variants can be used if they fail on a large number of input cases. This issue is further discussed below.

3.2 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 GP 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 [Saglietti90] and [Lyu94]). The taxonomy has been carried over from the Software Metrics area [Fenton91]. 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. Fenton arrives at this taxonomy by viewing a piece of software as a set of activities (processes) using resources to produce artifacts (products) [Fenton91]. In table 2, a diversity source with leading number 1 is a process, with leading number 2 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 GP 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, Testing scheme
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 [Fenton91], [Saglietti90] and [Lyu94]) 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 the GP programs should use. 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 [Lyu94]. 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 variants.

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.

3.3 Cost of using genetic programming

Developing one program variant in GP is an automatic process, once the GP system has been set-up. 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 done in parallel, and different runs can be made in parallel. 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-variant system. The initial cost for the GP approach may be higher, however; we may need to try parameter combinations 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, 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 variants 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. And cannot be debugged in the ordinary sense. The programs may 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. Similar approaches may be used when maintenance is performed on the N-variant system owing to, for example, changing requirements.

3.4 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 for 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 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. Description of experiment

We have used a genetic programming system to conduct two rounds of experiments. In the first round 80 program variants were developed from the same specification and in the latter round 320 additional programs were developed. In the following we discuss the results from the 80-variant experiment. Detailed results from the second round of experiments can be found in appendix III and the analyses are briefly described in section 6.5.

All programs were developed automatically by a custom developed system running on a SUN Enterprise 10000 with the Sun Solaris OS 2.5 and Java Development Kit 1.2. The GP system was run five times for sixteen different settings of parameters. The resulting eighty programs were subjected to the same 10000 test cases and their failure behavior analyzed to assess the failure diversity of the programs. Figure 1 gives a sketch of the experiment environment. Below we describe the target system, the GP system, the design of the experiment and the testing procedure. A more detailed description is given in appendix I.

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 airplane the program should make the aircraft come to a smooth stop. The requirements on a system like this can be found in [US Air Force86]. 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 [Christmansson98].

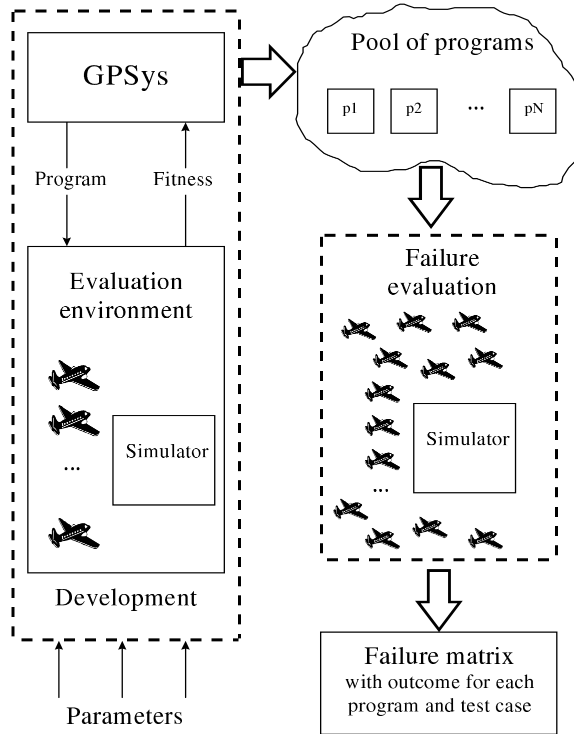

Figure 1. Experiment environment for developing and evaluating airplane arrestment controllers

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 having maximum energy of $8.81*10^7$ J and mass and velocity in the range 4000 to 25000 kg and 30 to 100 m/s, respectively. More formally the program should[1] (name of corresponding failure class in parentheses)

- stop aircraft at or as close as possible to a target distance (275 m)
- stop the aircraft before the critical length of the tape (335 m) in the system (OVERRUN)
- not impose a force in the cable or tape of more than 360 kN (CABLE)
- not impose a retarding force on the pilot corresponding to more than 2.8g (RETARDATION)
- not impose a retarding force exceeding the structural limit of the aircraft, given for a number of different masses and velocities in [USAF86] (HOOKFORCE)

[1] Our system adopts the requirements of [USAF 86] with the addition of the allowed ranges for mass and velocity and a critical length of 335 m (950 feet in [USAF 86]).

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

An existing simulator of the system has been ported from C to Java. It implements a simple mechanical model of the airplane and braking system and calculates the position, retardation, forces and velocities in the system. It does not model the inertia in 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 62.5 milliseconds.

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 the University College in London. The programs in this system are function trees, which 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 on 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 criteria:

- 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 criteria.
- 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 haven't 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 criteria.
- 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 ciritcal 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 to the system but is typically much smaller than 80%.

The penalties for the other criteria are assigned in a similar manner. For more details consult appendix I. 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 finesses on all the test cases. A perfect program would get 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 on the range of valid masses and velocities. Dividing the range of allowed mass into 100 location 212.12 kg 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 [USAF86]. The range [30, max velocity for this mass] is divided into 100 velocities and a total of 100*100=10000 test cases result.

4.4 Experimental design

The discussion in section 3.2 argued that the program space defining parameters (PSP) should have the largest effect on the diversity of the resulting programs. The parameters to the evaluation environment (EP) should also have an effect while the search parameters (SP) may primarily affect the effectiveness of the GP system. In accordance with this we have chosen to vary four program space parameters, three evaluation environment parameters and one search parameter. Many of these parameters can take multiple values giving rise to an enormous number of combinations. To make a study feasible we have confined the parameters to two levels, represented by '-' and '+'. The parameters and their levels are listed in table 2. More details on the choice of parameters and levels can be found in Appendix I. All other parameters to 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 2-valued parameters is 256 but we used a $2^{(8-4)}$ fractional factorial of resolution IV to reduce this to 16 [Box78] [USAF86]. The settings of the parameters are shown in table 3, where a '-' indicate the 'low' level of the parameter and a '+' indicate the 'high' level. Once the order in which to run the 80 experiments had been randomized the experiment was started. The system ran the 80 runs over a course of five days without any human intervention.

Factor	Level	Description
A	-	No effect.
	+	The statement IF, and operators LE, AND and NOT can be used in the programs.
B	-	No effect.
	+	The functions SIN and EXP can be used in the programs.
C	-	The average velocity, average retardation, and the index to the current checkpoint can be used in the programs.
	+	The angular velocity, current time since start of the braking, the previous angular velocity and the time of the previous checkpoint can be used in the programs.
D	-	Programs cannot use any subroutines.
	+	Two subroutines (automatically defined functions) can be used in the program. They are evolved in the same manner as the rest of the program.
E	-	Maximum penalty on the RETARDATION failure criteria is 1000.0.
	+	Maximum penalty on the RETARDATION failure criteria is 2000.0.
F	-	Linear penalties are not used.
	+	Linear penalties are used and a maximum penalty of 30.0 is assigned on each failure criteria.
G	-	25 test cases uniformly spread on the range of possible values for mass and velocity are used to evaluate fitness during evolution.
	+	25 test cases chosen randomly for each run of the GP system are used to evaluate fitness during evolution.
H	-	Probability of mutation is 0.05.
	+	Probability of mutation is 0.6.

Table 2. Factors that are varied in the experiment

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

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

5. Experimental results

For each test case executed, a trace of the braking of the airplane is returned from the simulator. Four values are extracted from this trace to classify the behavior 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 variant on a particular test case if *any* value exceeds its limits. Failure is indicated by one (1) and success by zero (0) and these binary values are collected into a failure behavior vector giving the failure behavior on a particular test case.

Setting	Run 1	Run 2	Run 3	Run 4	Run 5	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	2203	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	3146	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%

Table 4. The number of failures for each of the 80 versions, the average and the average success probability for each setting of the parameters

The quality of the eighty programs varies highly. Table 4 shows the observed failure rates of the variants. 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%) while 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.

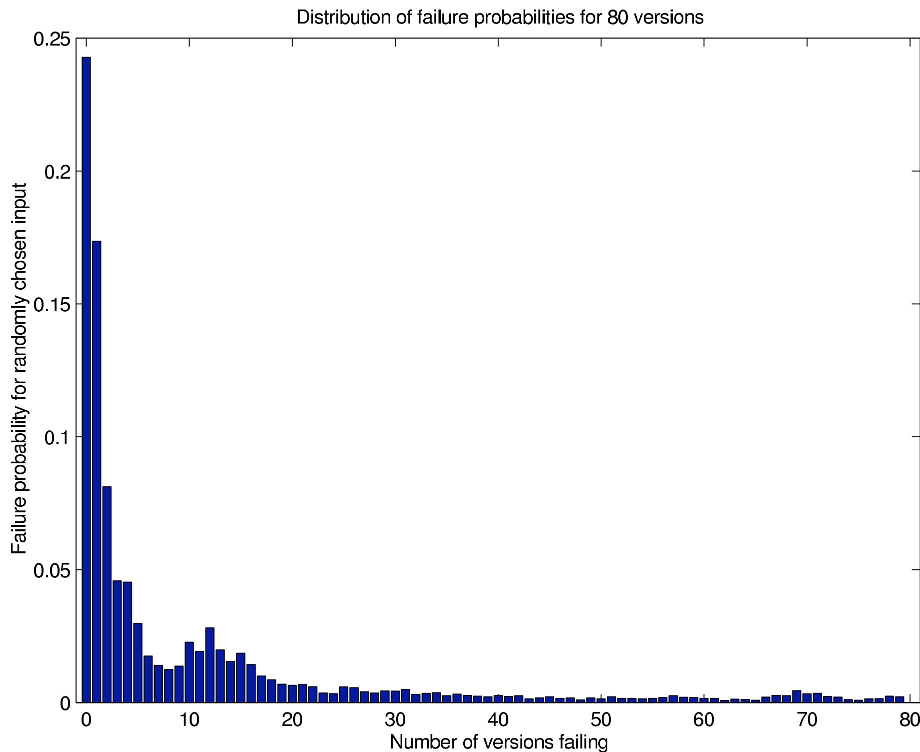

Figure 2. Distribution of failure probabilities for a randomly chosen input

Many programs failed on the same test case. Figure 2 shows the probability that n of the eighty variants fail on a randomly chosen test case among the 10000 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 figure 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 100.20 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 84.80 with a standard deviation of 46.07. The maximum size is 185 and the minimum is 38.

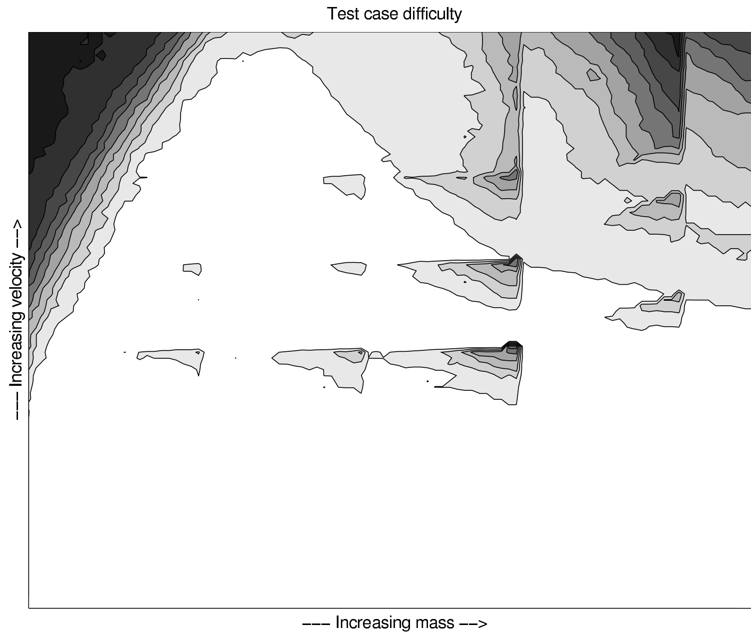

Figure 3. Contour plot of test case difficulty

6. Evaluation of results

Below we evaluate the failure diversity, test case difficulty variability and performance of 3-, 5- and 7-variant systems constructed from the programs and briefly describe the analysis of the second round of experiments with a total of 400 variants. 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 if varying the parameters to the system generates diversity.

6.1 Failure diversity

Different measures of diversity have been proposed in the literature. In [Littlewood89], Litttlewood 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 since 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 [Littlewood89]. Another failure diversity measure was used in [Lyu94]. It is defined as the number of distinct failures divided by the total number of failures, and below we denote it LFD.

Between programs. The diversity measures were calculated pairwise for all eighty

programs. The minimum correlation[2] was –0.2131 and of the 3160 correlations 193 (6.11%) were below zero. The maximum LFD was 0.9894. This is encouraging since low correlations and high failure diversity indicates that taking a vote among variants can mask 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.

Between methods. We have calculated the 120 inter-method correlations where each setting of the parameters to the GP system is considered a unique method. The majority of the correlations are high but 8 are below 0.20 and two are negative. This was surprising and indicates that the variability of difficulty of the test cases may be overcome and the program variants can show better than independent failure behavior. However, the majority of methods involved in the lowest inter-method correlations are the ones having highest average failure rate. Thus, even if we pick programs for N-variant systems from methods showing low correlation, the failure rate of the system will probably not level that of the top performing programs.

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

- Randomly choose one method (A) and two distinct programs (A1 and A2) from it
- Randomly choose a program not in A and call it B1
- Compare the diversity between A1 and A2 with the diversity between A1 and B1. 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 variants due to the different 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 10000 test procedures. For the correlation measure 6370 positive outcomes were recorded and for the failure diversity measure 6365 positive outcomes. 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 favor the hypothesis that *the failure diversity is larger between variants developed with different settings of the GP parameters than between variants 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[3]. In 10000 repetitions of the procedure 5613 (5551 with the failure diversity measure of Lyu et al) positive outcomes were recorded. Thus, the null hypothesis still could be rejected at the 0.01 level (both with p-value $< 10^{-10}$).

[2] Calculated as the correlation between the failure behavior vectors. This is a special case of the Littlewood and Miller correlation measure when there is only one variant in each method.

[3] Hence, methods number 4, 9, 10, 12 and 16 were excluded

6.2 Test case difficulty variability

Detailed study of the test case difficulty variability pictured in figure 3 reveals that there are three main areas of difficulty. Visually these areas is located in the upper left corner, in equidistant clusters in the center 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 center area are mainly due to failure on the HOOKFORCE criteria. The requirements in [USAF86] stated the maximum allowed hook force for certain "points" with specified mass and velocity. The clusters of failing programs seen in the center of figure 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 hook force. 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 HALTDISTANCE 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 in the start of the braking they will not have time to brake the aircraft before the critical length.

6.3 3-variant systems constructed from the programs

We constructed 3-variant systems from our programs. The majority vote between the failure behaviors 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 the 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 behavior (good performance early, failing in the end). With our post-run voting the behavior 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 the 120 possible N-variant systems consisting of 3 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%).

We also compared the performance of the best 3-variant systems to the performance of the best individual programs. Table 5 shows the top 25 programs or systems. In the table, the column marked 'Type' shows the type of system with 'Ind' indicating an individual program and '3VS' a 3-variant system. The column 'Failures' shows the number of failures. If the system is a '3VS' the column 'Improvement' shows the percent improvement in failure rate of the system compared to the best of the individual programs in the system.

As can be seen in the table no 3-variant system performed better than the best individual program. However, the 3-variant systems dominate the top list and only two individual programs perform good enough to qualify.

Rank	Type	Failures	Improvement
1	Ind	392	NA
2	3VS	444	11.02%
3	3VS	444	20.57%
4	3VS	444	20.57%
5	3VS	449	19.68%
6	3VS	455	18.60%
7	3VS	463	7.21%
8	3VS	464	16.99%
9	3VS	465	16.82%
10	3VS	467	16.46%
11	3VS	469	16.10%
12	3VS	470	5.81%
13	3VS	474	15.21%
14	3VS	484	3.01%
15	3VS	485	13.24%
16	3VS	490	12.34%
17	3VS	493	11.81%
18	Ind	499	NA
19	3VS	504	10.00%
20	3VS	515	7.87%
21	3VS	520	7.14%
22	3VS	524	6.26%
23	3VS	525	6.08%
24	3VS	525	6.25%
25	3VS	526	5.90%

Table 5. Top 25 performing individual programs and 3-variant systems

6.4 5- and 7-variant systems constructed from the programs

Five and seven variant systems was also constructed from the top 10 programs. No systems were found that improved upon the failure rate of the individual program in the system with the lowest failure rate.

6.5 Analysis of 400 variants

An additional 320 programs were developed in a second round of experiments. In this round, twenty programs were developed for each of the sixteen settings. The analyses above were carried out on the total of 400 programs and the detailed results can be found in Appendix III.

The results from these analyses are much the same as the ones given above with one notable exception. The negative and small inter-method correlations are not as frequent; only

one inter-method is below 0.20 (0.0785 between settings 9 and 14). This indicates that the smallest correlations for the 80 variant analysis may be sampling errors.

7. Discussion and conclusions

We have proposed a procedure for developing diverse software variants and shown that the variants 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-variants system could mask individual program failures. However, the methods giving the lowest correlations also are the ones with the highest failure rates, and the diversity cannot be exploited to give failure rates lower than the top performing programs. The diversity levels found in the top performing programs was much lower. Further analysis will be conducted to find out if the poor performance can be said to cause the high diversity.

The observed behavior may be explained by the special nature of the target system. It shows high level of input case difficulty variability, which is known to limit the amount of exploitable diversity [Littlewood89]. The difficulty arises from the fact that higher energies put more stress on the system. If this amount of input case variability is typical is not known. Further experiments with other target systems can shed light on this issue.

In our experiments we have varied eight parameters to 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 get to calculate the break pressure more frequently during a 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 behavior of programs we do not compare each failure criteria individually. Thus, diversity in the way the programs fail is not accounted for even though it might be exploited in a system employing fault masking. Our failure classification does not take the time aspect of the program behavior into account. A situation can easily be envisioned where two programs both exceed the maximum allowed hook force but at different times. This faulty behavior might be masked by an N-variant system. It would be interesting to actually construct N-variant systems from our programs and evaluate their failure behavior.

Further experimentation with the existing system will be conducted since it mainly amounts to initiating runs and collecting and analyzing data; the development of the programs requires no human activity. Having large numbers of software variants 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 [Avizienis95].

8. References

[Avizienis77] AVIZIENIS, A. and CHEN, L.: 'On the implementation of N-variant programming for software fault-tolerance during program execution', Proc. of COMPSAC-77, 1977, pp. 149-155

[Knight86] KNIGHT, J. C. and LEVESON, N.: 'An experimental evaluation of the assumption of independence in multivariant programming,' *IEEE Trans. on Software Engineering*, **12** (1) pp. 96-109

[Lyu 94] LYU, M., CHEN, J-H. and AVIZIENIS, A.: 'Experience in metrics and measurements for N-variant programming,' *Int. Journal of Reliability, Quality and Safety Engineering*, **1** (1) pp. 41-62

[Littlewood89] LITTLEWOOD, B. and MILLER, D. R.: 'Conceptual modelling of coincident failures in multivariant software,' *IEEE Trans. on Software Eng.*, **15** (12) pp. 1596-1614

[Hatton97] HATTON, L.: 'N-Variant design versus one good variant', *IEEE Software*, **14** (6) pp. 71-76

[Koza92] KOZA, J. R.: 'Genetic programming - on the programming of computers by means of natural selection' (MIT Press, Cambridge, Massachusetts, 1992)

[Bäck97] BÄCK, T., HAMMEL, U. and SCHWEFEL, H-P.: 'Evolutionary computation: comments on the history and current state,' *IEEE Trans. on Evolutionary Computation*, **1** (1) pp. 3-17

[Koza97] KOZA, J. R. (ed): 'Proceedings of Second Annual Conf. on Genetic Programming July 13-16, 1997' (Morgan Kaufmann, San Fransisco, California, 1997)

[Nordin95] NORDIN, P. and BANZHAF, W.: 'Real time evolution of behavior and a world model for a miniature robot using genetic programming', Technical Report 5/95, Department of Computer Science, University of Dortmund, 1995

[Ryan96] RYAN, C. O.: 'Reducing premature convergence in evolutionary algorithms.' PhD Dissertation, Computer Science Department, University College, Cork, 1996

[Banzhaf98] BANZHAF, W., NORDIN, P., KELLER, R. and FRANCONCE, F.: 'Genetic programming - an introduction' (Morgan Kaufmann, San Fransisco, California, 1998)

[Zhang97] ZHANG, B-T. and Joung, J-G.: 'Enhancing robustness of genetic programming at the species level', *Proc. of Second Annual Conference on Genetic Programming*, July 1997, Stanford University, USA, pp. 336-342

[Box78] 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)

[USAF86] 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

[Christmansson98] CHRISTMANSSON, J.: 'An exploration of models for software faults and errors', PhD Dissertation, Department of Computer Engineering, Chalmers University of Technology, 1998

[Avizienis95] AVIZIENIS, A.: 'Building dependable systems: how to keep up with complexity'. *Special Issue from FTCS-25 Silver Jubilee*, June 1995, Pasadena, California, pp. 4-15

[Saglietti90] SAGLIETTI, F.: 'Strategies for the acheivement and assessment of software fault-tolerance'. *IFAC 11th World Congress on Automatic Control*, Tallinn, USSR, 1990, pp. 303-308

[Fenton91] FENTON, N. E.: 'Software Metrics - A Rigorous Approach'. (Chapman & Hall, 1991)

Appendix I. Detailed description of the experiment environment

Below we give additional information on different parts of the experiment environment: the genetic programming system, Java system, the custom developed Java code, parameter values for the GP system, parameters that are varied in the experiment, design of the factorial experiment, fitness evaluation, the runs of the system and the analysis performed.

Genetic programming system

We have used versions 1.1 of the GP system called 'GPSys' developed by Adil Quereshi at the University College in London. This version of the system was released on the 30th of June 1997 and can be downloaded for free from the following web site:

http://www.cs.ucl.ac.uk/staff/ucacaxq/gpsys_doc.html

(if this URL is no longer valid you can contact the author of this paper to get a copy of the GPSys system).

GPSys is a strongly typed steady-state genetic programming system written in Java. It requires Java version 1.1 or later. The system is structured into a base package and a package of primitives, i.e. the terminals and functions that can be used in the development of programs. Both packages are object oriented and can be easily extended by writing additional Java classes. The system have support for automatically defined functions, i.e. sub-routines.

The major drawback of the system is that it is not compiled into a native machine language; the performance of the system is relatively poor.

Java system

We have used SUN's Java development kit 1.2 with the sunwjit just-in-time compiler. The first time a Java class is loaded it is compiled to machine code for the Sun Sparc architecture and can be executed with greater performance on following invocations. The speed-up achieved with the jit compiler was between 2-3 times.

Custom developed Java code

A total of 28 Java classes were developed to extend the GPsys system with additional functionality needed in the chosen application. These classes contain a total of 1724 lines of source code (comments excluded). Fifteen of the classes implement the evaluation of the programs by interfacing to the simulator defining test cases, assembling information about the braking, evaluating the braking and assign a fitness score. Thirteen classes interface with the GPSys system and the user. They are implemented so that the levels for the factors determining the parameter values that are varied can be given from the command line interface when an experiment is started.

Parameter values for the GP system

Eight different parameters were allowed to vary in the experiments. Table 6 below shows the values of the parameters that did not vary between different runs. When ADF were used two ADF's were allowed: ADF1 and ADF2. The values of parameters for these ADF's are shown in table 7 and 8, respectively.

Parameter	Value
Generations	200
Population size	1000
Tournament size	5
Max depth of program trees at creation	7
Max depth of program trees	19
Max depth of mutation trees	3
Create Method	Ramped-half-and-half

Table 6. Values of parameters that were not varied during the experiments

Parameter	Value
Max depth of program trees at creation	5
Max depth of program trees	9
Max depth of mutation trees	3
Functions allowed	Add, Sub, Mul, Div, If, GE, LE, ADF2
Terminals	Arg1, Arg2, Arg3, Double constant
Create Method	Ramped-half-and-half

Table 7. Values of parameters in ADF1

Parameter	Value
Max depth of program trees at creation	5
Max depth of program trees	9
Max depth of mutation trees	3
Functions allowed	Add, Sub, Mul, Div
Terminals	Arg1, Arg2, Double constant
Create Method	Ramped-half-and-half

Table 8. Values of parameters in ADF2

Factors that are varied in experiments

Of the eight factors varied in the experiment four (A, B, C and D) are program space parameters (PSP). Factors A and B allows the use of additional functions. When factor A is on its high level (indicated with a '+' in table 2) the programs can use the 'function' IF and the three operators LE (Larger-than-or-equal), AND (logical And), and NOT (logical negation). When factor B is on its high level the functions SIN and EXP can be used in the

programs. It was thought that these functions might give diverse algorithms since they are suited to model oscillatory and damping behavior, respectively. Factor C governs what terminals can be used by the programs and factor D governs if subroutines can be used. When it is on its high level two subroutines can be used in the programs. The subroutines are evolved together with the main program.

Three factors (E, F and G) are evaluation parameters (EP) affecting how the programs are evaluated. Factors E and F alters different aspects of the fitness evaluation while factor G affects the number and choice of test cases. When factor G is at its low level (indicated with '-' in table 2) the test cases are evenly distributed on the range of allowed values of mass and incoming velocity. When it is on its high level the values for mass and velocity are randomly chosen.

One factor (H) is a search parameter (SP) governing the probability of mutation in the GP system. Initial runs indicated that high values might be beneficial.

Design of the factorial experiment

The defining relation for the factorial experiment is [Box78]:

$$I = BCDE = ACDF = ABCG = ABDH$$

To generate the 16 different settings we listed all 16 combination of factor levels for the factors A, B, C and D. From the defining relations the values for the remaining factors was calculated according to

$E = B*C*D$
$F = A*C*D$
$G = A*B*C$
$H = A*B*H$

where a '+' is assigned the value '+1' and '-' the value '–1'. For more information on these issues see [Box78].

Fitness evaluation

When program evaluation takes place a program is tested on a number of test cases, i.e. air planes with a certain mass and velocity, coming in to land on a runway. An aircraft is characterized by its mass and incoming velocity. The braking is simulated in time steps and at each time step a number of data about the braking is logged. The data is the position, velocity and retardation of the aircraft, the force on the hook of the aircraft, the force in the cable, and the pressure applied on the tape drum by the braking system.

After each braking the logged data are analyzed to evaluate if any of the four success criteria have been violated. On each criteria a penalty value is assigned and these values are added to give the penalty value on the braking. Bu summing the penalty values for all the brakings an aggregated penalty value is obtained. The penalty value is used as an "inverse" fitness value so that high penalty indicates low fitness and low penalty indicates high fitness. A "perfect" individual has a penalty value of zero indicating that no success criteria were violated, i.e. the program adhered completely to the specification, on the test cases.

The halt distance criteria (OVERRUN) ensures that the aircraft were arrested before the critical length of 335 meters. If the program did not succeed, i.e. the airplane speed was larger than zero at the critical length of 335 meters, a basic penalty of 800 units is assigned.

An additional penalty of maximum 200 units is assigned linearly based on how much the velocity of the airplane was when it exceeded the critical length. The maximum value of 200 is added when the velocity at the critical length is the same as the incoming velocity, i.e. the system has not even tried to brake the system. When the system has been able to brake the aircraft before the critical length a small penalty is assigned based on how much the braking distance deviates from the target distance of 275 meter. This penalty increases from zero at the target distance to a maximum of 30 units at the critical length and at the engaging position. The penalty assignment on this criterion is shown in figure 4 below.

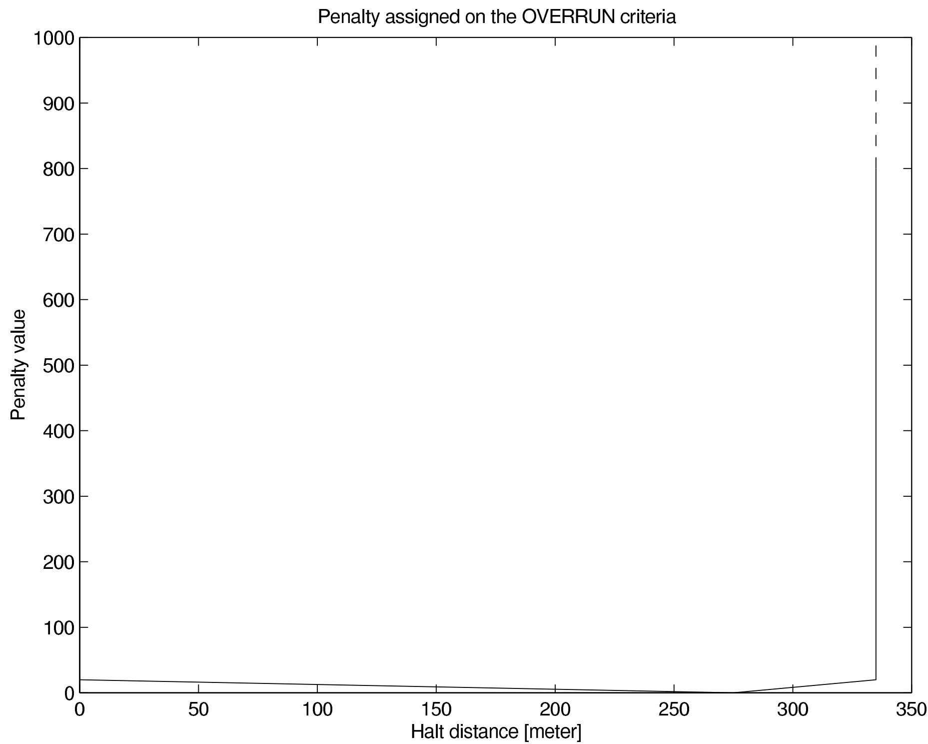

Fig 4. Penalty assignment on the OVERRUN criterion

The hook force criteria (HOOKFORCE) is parameterized on the maximum hook force that the aircraft can handle. These maximum hook forces are taken from the specification and can be found in a table in the document [USAF86]. The table gives the maximum allowed force in the hook on the aircraft with a specified mass and incoming velocity for a total of seventeen different mass and velocity "points". For masses and velocities not given in the table we have used the value of the closest given point that have the smallest higher mass and smallest higher velocity. This divided the mass and velocity plane into 17 boxes where the same maximum hook force is allowed. However, with the additional requirements we posed on the system that it should handle velocities up to 100 m/s and masses up to 25000 kg some parts is not covered by the 17 boxes. For these points we have used a formula obtained from the seventeen given values using linear regression.

If the maximum hook force exerted during a braking has been larger than the maximum hook force that the aircraft can handle, called the critical hook force, a basic penalty of 800 units is assigned. A linear extra penalty is assigned for maximum hook forces over the

critical hook force taking the maximum value of 200 units at two times the critical hook force. A small linear penalty is assigned that has a value of zero at a maximum hook force of zero and the value 30 units at the critical hook force. An example of a penalty assignment on this criterion, for a critical hook force of 220.2 kN for aircraft with masses in the range [13608, 18144] kg and velocities in the range [61.8, 72.1] m/s, is shown in figure 5 below.

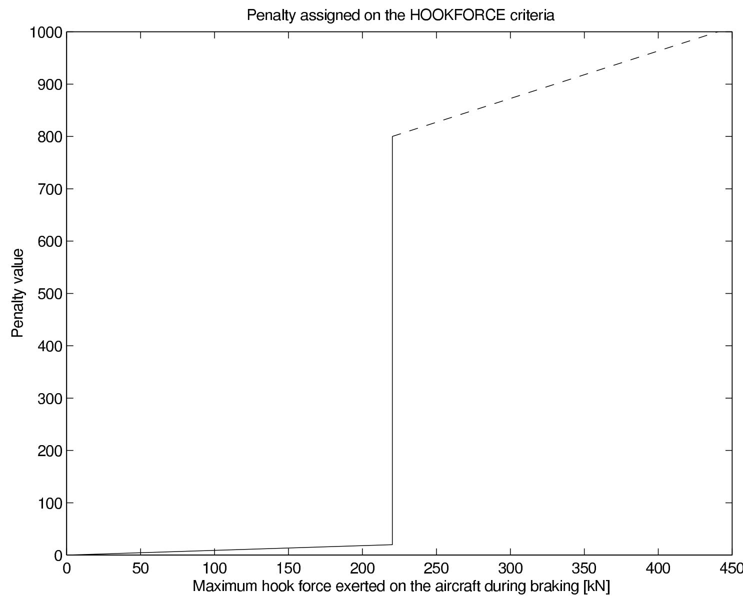

Fig 5. Penalty assignment on the HOOKFORCE criterion with a critical hook force of 220.2 kN

The penalty for the CABLEFORCE and RETARDATION criterion is assigned in the same way as the HOOKFORCE criterion above. The only differences is the critical value where the penalty value changes abruptly. For the CABLEFORCE criterion this critical value is the maximum allowed force in the cable that the braking system can handle. For the particular system used the value is 360.0 kN. For the Retardation criteria the critical value is 2.8*g, which is chosen to ensure that the pilot will not pass out during the braking.

Additional details about the runs

Each run of the development system has got a unique experiment number. The output from each run is four files: the experiment file, the failure data file, the log file and a file with the best of run individual.

The experiment file is a complete description of the experiment logging the experiment number, start time, duration of the run, the factor levels and the parameter values, fitness and complexity of the best-of-run individual, and a summary of the evaluation results.

The failure data file contains the result from the 10000 test cases. A number is output for each test case. The four low-order bits of the number corresponds to the four different failure classes and if a bit is '1' the best-of-run individual failed on these criteria for this particular test case.

In the log file data about each generation of the run is logged during the run. For each generation the average fitness, average complexity, best individual, fitness of best individual

and complexity of best individual is printed in the log. The log data can be used to analyze the evolutionary process.

The best-of-run individual is saved in binary form so that it can be loaded at a later time for additional tests.

Execution time of the runs

The execution time of the runs varied a lot. For the eighty programs the mean execution time was 19756 seconds, i.e. about 5 and a half hours. The standard variation was 9105 seconds, i.e. about 2 and a half hours. However, since each run can be run separately and the computer contained a total of fourteen CPU's the runs were parallellized and the experiments could be completed in a shorter time span.

The execution time is highly dependent on the use of the Java interpreting programming language. Using a GP system implemented in C och C++ or even a GP system using machine language for the program representation should decrease the execution times considerably. Nordin have reported speed-up factors of over 1000 times using a machine language representation [Banzhaf98]. Using an environment like that of Nordin would allow larger populations and more generations which should increase the likelihood of finding good solutions with small failure rates.

The process of starting experiment runs was automated so the process could be initiated and then left on its own.

Analysis of experiments

Analysis are performed off-line using MATLAB. Scripts have been written for the analyses and once all the experiment numbers have been listed and been read into MATLAB the scripts performs the analyses. The results are written to an output file and figures are generated from MATLAB and written on files. An example of such a file is shown in appendix II and III below.

Appendix II. Detailed results from the analysis of 80 variants

The analysis of the failure behaviors was carried out in MATLAB. A MATLAB-script runs the analyses and outputs the results in textual form to a file. Below the resulting file from the analyses of the 80-variant experiment described in section 4 and 5 are shown.

Tests and analyses for technical report 98-13 on the GPBRExperiments using 80 versions
----
Average number of failures: 1298.96 (Psucc=87.01%)
Standard deviation: 712.89
Best program, number of failures: 392 (Psucc=96.08%)
Worst program, number of failures: 3609 (Psucc=63.91%)
Top 10, Average number of failures: 553.90 (Psucc=94.46%)
Top 10, Standard deviation: 65.93

Graph showing probability that n versions out of 80 versions
fail on a randomly chosen test case among the 10000 test cases was written to file

fig_probfailure.eps

Number of testcases that fail on 80 versions: 0
Number of testcases that fail on 79 versions: 22
Number of testcases that fail on 78 versions: 24
Number of testcases that fail on 77 versions: 15
Number of testcases that fail on 76 versions: 14
Number of testcases that fail on 75 versions: 9
Number of testcases that fail on 74 versions: 11
Number of testcases that fail on 73 versions: 21
Number of testcases that fail on 72 versions: 23
Number of testcases that fail on 71 versions: 35
Number of testcases that fail on 70 versions: 33

Contourplot of testcase difficulty variability written to file

fig_testcase_difficulty.eps

Structural diversity

Average size: 100.20
Standard deviation: 82.87
Max size: 459
Min size: 17

Correlation between size and failure rate: 0.05

Top 10 programs
Average size: 84.80
Standard deviation: 46.07
Max size: 185
Min size: 38

Failure diversity

1, Between programs

All programs:
Minimum correlation: -0.2131
Number of correlations: 3160
Negative correlations: 193.00 (6.11%)
Small correlations (<0.2): 574.00 (18.16%) Note that the negative correlations are included in the small correlations.
Maximum failure diversity: 0.9894

Top programs:
Minimum correlation: 0.5495
Maximum failure diversity: 0.5965

2, Between methods

Number of intermethod correlations: 120
Number of negative correlations: 2
Number of small correlations (<0.2): 8 Note that the negative correlations are included in the small correlations.

The 20 smallest inter-method correlations:

Method 1	Method 2	Correlation
9	14	-0.0454
9	11	-0.0180
9	12	0.0766
1	9	0.0978
7	9	0.0994
8	9	0.1322
4	14	0.1685
9	13	0.1869
4	11	0.2604
3	9	0.2758
9	10	0.2887
4	12	0.3080
2	9	0.3224
1	4	0.3703
4	7	0.3754
6	9	0.3832
5	9	0.4227
4	8	0.4367
9	16	0.4735
5	14	0.4963

3, Inter- vs. Intra

All programs:

The number of possible binomial tests: 24000
Number of tests performed: 10000

Correlation measure gave 6370 positive outcome.

Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 6365 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than average:

Methods worse than average that are NOT included: 4 9 10 12 16
Methods better than average: 1 2 3 5 6 7 8 11 13 14 15
Number of methods used in the tests below: 11
The number of possible binomial tests: 11000
Number of tests performed: 10000

Correlation measure gave 5613 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 5551 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than second average (average of the ones that were better than average above):

Methods worse than second average that are NOT included: 1 2 3 4 6 9 10 11 12 14 15 16
Methods better than second average: 5 7 8 13
Number of methods used in the tests below: 4
The number of possible binomial tests: 1200
Number of tests performed: 1200

Correlation measure gave 718 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=5.3169e-12)
Diversity measure gave 695 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=2.2891e-08)

Construct 3VP systems from the top 10 programs

Number of 3VS that can be constructed: 120
Number of 3VS that was constructed: 120
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 41 (34.17%)
Best improvement:
minimum failures of individual program = 559
failures of 3VS = 444
percent improvement = 20.57%

Construct 5VP systems from the top 10 programs

Number of 5VS that can be constructed: 252
Number of 5VS that was constructed: 252
Number of 5VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
minimum failures of individual program = 559
failures of 3VS = 620
percent improvement = -10.91%

Construct 7VP systems from the top 10 programs

Number of 7VS that can be constructed: 120
Number of 7VS that was constructed: 120
Number of 7VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
minimum failures of individual program = 559
failures of 3VS = 711

percent improvement = -27.19%

Combined top list of individual program or 3VS among the top programs
----

Rank	Type	Prg 1	Prg 2	Prg 3	Failures	Improvement
1	Ind	747			392	NA
2	3VS	744	758	827	444	11.02%
3	3VS	758	827	803	444	20.57%
4	3VS	758	827	770	444	20.57%
5	3VS	801	758	827	449	19.68%
6	3VS	758	827	743	455	18.60%
7	3VS	744	801	758	463	7.21%
8	3VS	758	827	776	464	16.99%
9	3VS	758	827	764	465	16.82%
10	3VS	801	758	803	467	16.46%
11	3VS	801	758	764	469	16.10%
12	3VS	744	801	827	470	5.81%
13	3VS	801	827	803	474	15.21%
14	3VS	744	758	743	484	3.01%
15	3VS	758	803	743	485	13.24%
16	3VS	758	764	743	490	12.34%
17	3VS	801	827	764	493	11.81%
18	Ind	744			499	NA
19	3VS	827	803	743	504	10.00%
20	3VS	801	758	776	515	7.87%
21	3VS	827	764	743	520	7.14%
22	3VS	758	764	770	524	6.26%
23	3VS	758	803	770	525	6.08%
24	3VS	827	803	770	525	6.25%
25	3VS	801	758	770	526	5.90%

Construct 3VP systems from ALL programs

Number of 3VS that can be constructed: 120
Number of 3VS that was constructed: 82160
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 14818 (18.04%)
Best improvement:
minimum failures of individual program = 1746
failures of 3VS = 1010
percent improvement = 42.15%
Total time needed for analysis 83.7 (minutes)

Appendix III. Detailed results from the analysis of 400 variants

The original experiment was extended by developing an additional of 320 variants, 20 for each of the 16 settings of parameters. The result from the analysis carried out in MATLAB is shown below.

Tests and analyses for technical report 98-13 on the GPBRExperiments using 400 versions
----
Average number of failures: 1308.96 (Psucc=86.91%)
Standard deviation: 680.65
Best program, number of failures: 392 (Psucc=96.08%)
Worst program, number of failures: 3701 (Psucc=62.99%)
Top 10, Average number of failures: 463.40 (Psucc=95.37%)
Top 10, Standard deviation: 35.60

Graph showing probability that n versions out of 400 versions
fail on a randomly chosen test case among the 10000 test cases was written to file

fig_probfailure.eps

Number of testcases that fail on 400 versions: 0
Number of testcases that fail on 399 versions: 0
Number of testcases that fail on 398 versions: 0
Number of testcases that fail on 397 versions: 0
Number of testcases that fail on 396 versions: 0
Number of testcases that fail on 395 versions: 0
Number of testcases that fail on 394 versions: 1
Number of testcases that fail on 393 versions: 0
Number of testcases that fail on 392 versions: 1
Number of testcases that fail on 391 versions: 13
Number of testcases that fail on 390 versions: 9

Contourplot of testcase difficulty variability written to file

fig_testcase_difficulty.eps

Structural diversity

Average size: 108.01
Standard deviation: 97.06
Max size: 725
Min size: 5

Correlation between size and failure rate: 0.04

Top 10 programs
Average size: 107.20
Standard deviation: 49.44
Max size: 226
Min size: 59

Failure diversity

1, Between programs

All programs:
Minimum correlation: -0.4438
Number of correlations: 79800
Negative correlations: 4518.00 (5.66%)
Small correlations (<0.2): 13708.00 (17.18%) Note that the negative correlations are included in the small correlations.
Maximum failure diversity: 0.9995

Top programs:
Minimum correlation: 0.5371
Maximum failure diversity: 0.6150

2, Between methods

Number of intermethod correlations: 120
Number of negative correlations: 0
Number of small correlations (<0.2): 1 Note that the negative correlations are included in the small correlations.

The 20 smallest inter-method correlations:

Method 1	Method 2	Correlation
--------	--------	-----------
9	14	0.0785
9	12	0.2050
1	9	0.3000
7	9	0.3182
4	14	0.3184
9	11	0.3258
3	9	0.4309
8	9	0.4354
4	12	0.4510
5	14	0.4938
14	16	0.4956
14	15	0.4969
9	10	0.5011
9	13	0.5013
12	16	0.5685
1	4	0.5713
4	7	0.5770
5	12	0.5911
13	14	0.6076
2	9	0.6083

3, Inter- vs. Intra

All programs:

The number of possible binomial tests: 3600000
Number of tests performed: 10000

Correlation measure gave 6318 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

Diversity measure gave 6312 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than average:

Methods worse than average that are NOT included: 4 6 9 10 12 14 15 16
Methods better than average: 1 2 3 5 7 8 11 13
Number of methods used in the tests below: 8
The number of possible binomial tests: 840000
Number of tests performed: 10000

Correlation measure gave 5897 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 5949 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than second average (average of the ones that were better than average above):

Methods worse than second average that are NOT included: 2 3 4 5 6 8 9 10 12 14 15 16
Methods better than second average: 1 7 11 13
Number of methods used in the tests below: 4
The number of possible binomial tests: 180000
Number of tests performed: 10000

Correlation measure gave 6313 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 6506 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

Construct 3VP systems from the top 10 programs

Number of 3VS that can be constructed: 120
Number of 3VS that was constructed: 120
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 32 (26.67%)
Best improvement:
    minimum failures of individual program = 488
    failures of 3VS = 405
    percent improvement = 17.01%

Construct 5VP systems from the top 10 programs

Number of 5VS that can be constructed: 252
Number of 5VS that was constructed: 252
Number of 5VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
    minimum failures of individual program = 462
    failures of 3VS = 479
    percent improvement = -3.68%

Construct 7VP systems from the top 10 programs

Number of 7VS that can be constructed: 120
Number of 7VS that was constructed: 120
Number of 7VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
    minimum failures of individual program = 462
    failures of 3VS = 516
    percent improvement = -11.69%

Combined top list of individual program or 3VS among the top programs

----

Rank	Type	Prg 1	Prg 2	Prg 3	Failures	Improvement
1	Ind	747			392	NA
2	3VS	1140	1070	1042	394	5.97%
3	3VS	1140	1017	1042	395	5.73%
4	3VS	1140	1029	1042	396	5.49%
5	3VS	1140	1042	744	397	5.25%
6	3VS	1140	1042	1189	397	5.25%
7	3VS	1017	1042	1134	403	13.33%
8	3VS	1070	1042	1134	404	12.93%
9	3VS	1029	1042	1134	405	12.34%
10	3VS	1042	1134	744	405	17.01%
11	3VS	1042	1134	1189	405	17.01%
12	Ind	1140			419	NA
13	3VS	880	1029	1042	422	6.43%
14	3VS	880	1070	1042	423	6.21%
15	3VS	880	1017	1042	424	5.99%
16	3VS	880	1042	744	425	5.76%
17	3VS	880	1042	1189	425	5.76%
18	3VS	880	1042	1134	427	5.32%
19	3VS	1070	1017	1134	443	4.53%
20	3VS	880	1070	1017	445	1.33%
21	3VS	1070	1017	1042	446	3.88%
22	3VS	1029	1070	1134	451	2.38%
23	3VS	1029	1017	1134	451	2.38%
24	Ind	880			451	NA
25	3VS	1029	1070	1042	452	2.16%

Construct 3VP systems from the 100 best programs

Number of 3VS that can be constructed: 161700
Number of 3VS that was constructed: 161700
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 51799 (32.03%)
Best improvement:
minimum failures of individual program = 639
failures of 3VS = 415
percent improvement = 35.05%

Total time needed for analysis 238.5 (minutes)