Automatic control program creation using concurrent Evolutionary Computing

John K. Hart

A thesis submitted in partial fulfilment of the requirements of Bournemouth University
for the degree of Doctor of Philosophy

January 2004

Abstract

Over the past decade, Genetic Programming (GP) has been the subject of a significant amount of research, but this has resulted in the solution of few complex real – world problems. In this work, I propose that, for some relatively simple, non safety – critical embedded control applications, GP can be used as a practical alternative to software developed by humans.

Embedded control software has become a branch of software engineering with distinct temporal, interface and resource constraints and requirements. This results in a characteristic software structure, and by examining this, the effective decomposition of an overall problem into a number of smaller, simpler problems is performed. It is this type of problem amelioration that is suggested as a method whereby certain real – world problems may be rendered into a soluble form suitable for GP.

In the course of this research, the body of published GP literature was examined and the most important changes to the original GP technique of Koza are noted; particular focus is made upon GP techniques involving an element of concurrency - which is central to this work. This search highlighted few applications of GP for the creation of software for complex, real – world problems – this was especially true in the case of multi – thread, multi - output solutions.

To demonstrate this idea, a concurrent Linear GP (LGP) system was built that creates a multiple input - multiple output solution using a custom low – level evolutionary language set, combining both continuous and Boolean data types. The system uses a multi – tasking model to evolve and execute the required LGP code for each system output using separate populations: Two example problems – a simple fridge controller and a more complex washing machine controller – are described, and the problems encountered and overcome during the successful solution of these problems, are detailed. The operation of the complete, evolved washing machine controller is simulated using a graphical LabVIEW application.

The aim of this research is to propose a general – purpose system for the automatic creation of control software for use in a range of problems from the target problem class - without requiring any system tuning: In order to assess the system search performance sensitivity, experiments were performed using various population and LGP string sizes; the experimental data collected was also used to examine the utility of abandoning stalled searches and restarting. This work is significant because it identifies a realistic application of GP that can ease the burden of finite human software design resources, whilst capitalising on accelerating computing potential.

Related Publications:

Hart J., Shepperd M., Evolving software with multiple outputs and multiple populations. Late breaking Papers. Proceedings of Genetic and Evolutionary Computing Conference (GECCO) New York 2002, Morgan Kaufmann Publishers.

Hart J. Shepperd M. The evolution of concurrent control software using genetic programming. EuroGP2004, Coimbra, Portugal. European conference on Genetic Programming proceedings. Springer-verlag Berlin Heidelberg.

Acknowledgements

I wish to thank my primary supervisor, Professor Martin Shepperd, who gave me the opportunity to pursue this research in a subject of direct interest to me. I also wish to thank Martin for the unusual amount of time and effort he has set aside in his supervision here.

I also wish to thank my second supervisor, Dr Martin Lefley, for supplying an essential alternative view on this research.

Contents

Chapters:

1.	Introduction	6
2.	Literature search	13
3.	Control Systems	38
4.	The experimental system	48
5.	Experimentation	64
6.	Conclusion	85
	References	93

Appendices:

A. Evolutionary language set

B. Gaussian random number generation

C. Fitness of random strings at creation

1 Introduction

1.1 Introduction

This chapter highlights technological and social changes that are simultaneously easing software development whilst the demand for software – and software practitioners - grows: Within this context, evolutionary computing is introduced as a possible means for creating software, and a target user group of electrical product designers / engineers for such a system, is proposed.

1.2 The accelerating demand for software

The main aim of this research is to propose and demonstrate a method of automatic software creation for certain embedded control problems using concurrent Evolutionary Computing (EC). This is important because the method can capitalise on growing computer power in a context of increasing software demand and finite staff.

The continued advances in integrated circuit design, and materials science in general, has led to increasing speed, complexity and capacity of computer hardware, whilst reducing power requirements, size and cost: This trend is expected to continue until a molecular size limit is hit[1] - but the growth in parallel computing will maintain the momentum in many of these areas. In addition, the introduction of the microprocessor and Field Programmable Gate Array (FPGA)[2] have facilitated generic hardware design, and hence the mass demand and production of these devices, leading to further cost reduction.

---

[1] Moore predicted that available computing power will continue to double approximately every eighteen months due to increases in integrated circuit capacity: Intel suggest that this trend will continue until the end of the decade [http://www.intel.com/research/silicon/mooreslaw.htm].

[2] FPGAs:
Field Programmable Gate Arrays are dynamically reconfigurable, single chip devices typical comprising an uncommitted array of simple logic, various latched devices and RAM. The connectivity between the device elements and the chip outputs (and hence the overall FPGA function) is determined by the bit pattern written into the RAM. The bit pattern is generated by an optimising compiler using information supplied by the designer in the form of graphical schematics, algorithms (programs) or truth tables. For more information: http://xilinx.com]

The continued infiltration of cheap and increasingly ubiquitous computing into our lives would then seem to be inevitable, and this will naturally lead to a corresponding demand for software driven by the following factors:

- Software for new products

  The availability of compact, powerful, low cost, low power computing will enable the continued introduction of new products that would never have been a practical proposition before; The ideal example here is the mobile 'phone.

- Software for upgraded products

  Market competition to add new features and generally improve products will force computing into existing products. In addition to cost savings, incorporating the type of electronic devices outlined above can reduce engineering effort associated with re-design and maintenance: For example, it is far simpler to re – work the software or FPGA – based hardware, than re – engineer mechanical controllers for dishwashers or engine ignition.
  Further to this, the demand for local wireless networking such as BlueTooth or IEEE 802.11 (Wi-Fi)[3] promises to place computing in even the most commonplace of products.

- New software products

  New software products will emerge that need to be powered by fast, high capacity computers with good communication links: Computer games and image processing software are good examples of this.

---

[3] BlueTooth and IEEE 802.11 (Wi-Fi):
BlueTooth is a joint venture between Nokia and Ericsson and is a complete local radio network specification that has been released as an open source document. The advocates of BlueTooth anticipate the wholesale adoption of local radio networking. Further information at: http://www.bluetooth.com.
IEEE 802.11 is similar local wireless networking protocol – offering higher data rates than BlueTooth amongst other differences. For further information: http://www.weca.net/opensection/index.asp }

- Revision of existing products

    Better computer resources will similarly create an opportunity for upgraded versions of existing software to exploit. The resources consumed by the Windows operating system, for example, appears to have tracked the hardware available.

Unfortunately, this growth in software demand has outpaced the growth in software staff available with many wealthier countries importing software staff or exporting software development: If the staffing situation becomes more acute, then this may force companies developing software to consider the type of work offered to computer staff – during and after recruitment – if possible: Ideally, the software design work offered to humans would always be creative and challenging.
Fortunately, this growth in cheaper computing resources has allowed possibly inefficient or sub – optimal solutions to be developed for some applications i.e. the performance of the human computer user has not really increased: Thus applications such as word processors can now be written in high level languages (with a consequent reduction in development effort) rather than optimised in assembler for speed and code size.

Considering the current state of computing outlined above, namely; a rising demand for software, ever more useable computer resources, and a shortage of software staff; an opportunity for practical, automatic methods of software creation has arisen. In this research, Evolutionary Computing is harnessed to achieve this objective.

1.3 Evolutionary Computing

Genetic Programming (GP) [Koza 1992a] has been a research interest since the early 1990s, and can be viewed as an extension of Genetic Algorithm research [Holland 1975]. Both techniques have many similarities with the earlier work on Evolutionary Strategies [Rechenberg 1973] and even the first evolutionary computing work of Friedberg in the 1950s [Friedberg 1958]. Overall these methods can be classed as Evolutionary Computing, and typically use ideas that have analogies with the adaptation and development of natural biological

populations to their environments. Within these populations, the goal of the species is to carry the best genes (that encode the most suitable individuals) forward into the future generations of the population.
In GP, the solution will typically be a single, evolved program chosen from a population of programs created randomly for the first generation: The chosen individual here will be the program whose operation most closely matches the desired operation - as defined by the data mapping in the training set used to direct evolution.
GP is believed to be a useful search tool, most immediately perhaps, because it produces solutions that are ideal for the target medium – a computer. In action, GP can perform an efficient parallel search that both explores and exploits the search space: Exploration happens with the random sampling of the search space by the use of probabilistic adaptation of current population members; and Exploitation of the search space occurs by the identification of a tractable fitness landscape that can be followed to a suitable solution: GP techniques have already been successfully applied in the solution of practical problems [Popp et al 1998][Nyongesa et al 2001].

1.4 Target users

This PhD introduces the idea of using GP techniques to evolve software as an alternative to total human development; This is achieved with the prior use of problem decomposition into tractable code units. As detailed later, this decomposition is achieved by considering problem output requirements, and such information will be known to the target users.
This approach is not intended for general use on all problems – instead a specific subset of realistic target problems is identified. Similarly, this approach will necessarily target a suitable class of users who are able to perform this necessary decomposition.
Typical embedded control approaches are described and these will be seen to be shaped by the hardware output requirements of such software problems. Target users then, are the product designers and engineers responsible for adding embedded control to such products and, as such, will have the technical expertise needed to define the problem in terms of controller output. By adopting this technique, these product designers may be freed from the need to become practitioners in embedded control application in terms of programming or hardware design; and the alternative – the expensive and

possible unreliable outsourcing of such work – can be avoided. Similarly, later maintenance or re – design could involve re – running the system with a new training set.

1.5 Research aims and objectives

The aim of this research is to propose and demonstrate a method of automatic software creation for a subset of embedded control problems, using concurrent EC.

The objectives of this research are as follows:

- To examine the published GP research literature in order to establish whether any general trends exist: Particular attention will be paid to work relating to the use of parallel or concurrent methods in the creation of practical application software.

- To examine the existing practice in embedded control in order to highlight distinct features in this type of software construction that could be advantageous in automatic software creation. This will facilitate the identification of suitable bounds within which GP can be realistically used for software creation.

- To develop an EC – based system capable of creating a defined subset of control applications using a finite, universal language set that facilitates the majority of software action, in a simplistic, efficient form.

- To demonstrate the system above by the production of a solution for a realistic, relatively complex control problem with multiple outputs and data types.

- To experimentally examine the operation and sensitivity of the EC system developed in order to determine the scope of applicability i.e. will the system need to be tuned for each new problem type?

- To develop a visualisation tool to demonstrate the operation of the evolved control application.

- To suggest further topics for research that will add to this work in the areas of outstanding work, widening the scope of system applicability and improving the system performance.

1.6 Document map

The remaining chapters of this document are organised as follows:

Chapter 2: Literature search

Popular computational search methods are outlined here, but the main focus of this chapter is GP: The major trends in GP research are sought, along with significant changes to the original GP technique of Koza – especially where these are believed to be directly relevant to the automatic construction of practical programs for real application.

Chapter 3: Control systems

The temporal and structural hierarchies that make many embedded control programs distinct from other types of software, are described here: Such an approach to software construction (and hence problem decomposition) is proposed as potentially beneficial in EC software creation. Finally, the realistic operational bounds for such a creation technique (suggesting a target problem class) are outlined.

Chapter 4: The experimental system

Details the EC system used to implement the proposed software creation scheme: This includes information on the EC approach, language set, data types and parameters used. Two demonstration problems are described; the first for a simplistic fridge controller and the second for a relatively complex washing machine controller.

Chapter 5: Experimentation

The problems encountered, and the solutions adopted, during the development of the EC software creation system are described here: These include problems of premature convergence, appropriate fitness assessment and improving solution accuracy. The sensitivity to population size and program length - with regard to search performance - is experimentally examined, along with strategies for search termination and restarting. Lastly, a graphical simulation of the entire washing machine controller and OS operation is described.

Chapter 6: Conclusion

Here the completed work is examined, related to the research objectives and the contributions to knowledge detailed. This leads on to the suggested subjects for future research work.

2 Literature search

2.1 Introduction

This chapter outlines the various methods of metaheuristic computational search and then focuses in on Genetic Programming (GP) – which is central to this research: The body of published literature relating to GP is examined and categorised into one of three general types in an attempt to perceive trends in GP research. In addition, major changes to the original GP technique are highlighted, along with enhancements that may increase the real – world applicability of the technique; and the intention is to place this work in that context.

2.2 Metaheuristic Search and computational optimisation

All searches attempt to find something; in computational search the target may be the best solution in a search space consisting of possible solutions. Normally the method of obtaining a suitable solution would be to design it – if this is possible.

An alternative to design may be to reformulate the problem as a search problem and, in many cases, this may be a more appropriate approach for complex, combinatorial problems. Clarke [Clarke et al 2003] actively promote metaheuristic search as an alternative technique that can span the software engineering cycle; suggesting that as software engineering is actually engineering, then the real task involves satisfying many parallel constraints - rather than finding a single solution: Software engineering can thus be seen as suitable for metaheuristic approaches because the task will involve competing constraints, a large search space and cost function complexity. (Clarke does not include the actual creation of software amongst the suggested software engineering task targets).

As heuristic techniques, metaheuristic computational searches do not guarantee to yield the optimum solutions and are not necessarily repeatable processes due to their probabilistic nature. Applications for such optimization techniques have included financial forecasting, cost estimation, process scheduling and other forms of complex optimization problems. They are especially valuable where the problem at hand cannot be easily specified and so coded for conventional algorithmic solution i.e. by standard computation, or where the combinatorial possibilities make the search unfeasible by that method. Hillclimbing, Simulated Annealing (SA), Tabu Search (TS), Genetic Algorithms (GA) and Genetic Programming (GP) are all seen as metaheuristic search techniques that embody the above description.

2.3 Hillclimbing

Hillclimbing is a method of seeking the optimum solution to a problem by incrementally advancing the focus of the search over the search space (from a random starting point initially) towards an improved solution. The 'next move' is to be found by examining the values of solutions in the immediate neighbourhood of the current position, and then progressing to a better position - should one exist. This important region that can be considered as the 'neighbourhood' is determined by the operations available and the problem representation or coding used. In this way, the hill climbing method can be seen to rove the search space, moving upward to a maximal or optimum solution (or downhill to a minima if sought) provided that the search space is continuous. The incremental scope of such a technique is likely to be thwarted by any discontinuities encountered, and also may become trapped in local optima, unable to reach a possible global optimum. Hillclimbing is a very basic technique, but still suitable for certain problems where the search space landscape is amenable i.e. unbroken and without extremes of gradient. A current application of hillclimbing lies in image processing [Bunyaratavej & Miller 2002]: Here, HC is used to optimise image encoding whilst minimising the creation of digital noise in an environment where the 'cooling' effect of SA [see 2.4 below] is seen as too computationally intensive.

2.4 Simulated Annealing

Simulated annealing is a strategy that can be employed to improve upon basic hillclimbing or similar techniques. The process is analogous to that used to change the molecular structure of metals (and other materials) by heating, followed by slow cooling, which results in the annealed material becoming softer. In the computational equivalent, the heating and slow cooling is simulated by loosening the restrictions on the next move during the climb, i.e. to allow worse solutions for a number of iterations. As the search progresses the likelihood of accepting a poorer move is gradually reduced (the process 'cools'). The resulting search will have a wide search neighbourhood initially (which may ultimately allow for the discovery of a better overall solution) before focussing in on any optimum solution found.

Simulated annealing has been recently used to tune the coefficients for closed – loop motor speed control directly, either on or off – line, without resorting to a mathematical model of the target system [Acarnley & Al-Sadiq 2002]. SA is particularly appropriate here because this technique is capable of spanning the search space effectively and so cope with the discontinuities inherent in the drive / motor / load combination.

2.5 Tabu Search

With Tabu search [Glover 1990] the next move across the search space from the current position is made by considering the move against a list of moves that are deemed 'Tabu'. The forbidden or Tabu moves are collected as the search continues, using both short and long term memory of the search progress. The result can be a more efficient search, requiring fewer moves and generally less computational effort as knowledge about the characteristics of the search space are gathered. As part of strategies to escape local optima without losing any global optimum found, the Tabu restrictions are sometimes relaxed.

2.6 Evolutionary Search

Both SA and Tabu search can be added to other search techniques and both will give the search a better chance of achieving better results than, for instance, HC alone. This is a feature that can be seen to be inherent in evolutionary search techniques like Genetic algorithms[Holland 1975] and Genetic programming[Koza 1992]: A search using Genetic Algorithms is distinct from the above techniques in that it employs a population of candidate solutions that effectively perform the parallel sampling of the search space. Additionally, the use of a population allows GA to sample disjoint regions without the loss of any search progress already made.

GAs are classed as an evolutionary computing technique because the population can be seen to evolve so that the best - or most fit - solution to the problem at hand, will come to dominate the population, while less fit individuals disappear. Consequently, the Darwinian ideas of speciation and

extinction in evolution are frequently used in the description of these search methods.
The notion of search by evolutionary methods is not new: At the inception of electrical computing Turing (according to Bennett [Bennett et al 1999]) suggested that all intellectual activity consists of a mix of the three types of search: cultural (knowledge - based), genetically - based or a search through the number space.
The evolutionary operations of Selection, Crossover and Mutation are common to both GA and GP, though the detail reflects the differences in the operating environments. In GA the population is made up of a number of chromosomes that are formed by representing the parameters of the system (genes) together as a continuous binary string. This abstraction of the system is usually referred to as the phenotype and the system represented by it as the genotype.
In both GA and GP, population convergence can be used to signal an appropriate time to abandon further searching: In this thesis, the term Convergence is used to describe a situation whereby at least ninety – five percent of the population members are identical, and that this situation has persisted for at least the previous fifty generations. Here, the term Premature Convergence is used to denote a situation whereby the population has converged upon a solution - but a better solution is known to exist. This premature search stall upon a sub – optimal solution generally indicates a failure in the search method that allows the system to become trapped and unable to make further progress. Convergence is a condition whereby the range and variety of genetic material in the population is approaching a diversity minima: In this thesis, the term Diversity is used to refer to the variety in the population members: Thus a maximally – diverse population will contain only unique individuals whose differences cover the whole individual i.e. in both content and structure (in the case of GP). A highly diverse population will have a better chance of containing the components initially to build an ideal solution.

2.6.1 Evolutionary search: Genetic Algorithms

The Genetic Algorithm search [Holland 1975] proceeds in discrete generational stages, with each succeeding generation being formed by selection from the current generation of chromosomes. This selection process, along with the other operations and the creation of the initial population, are generated probabilistically.

The selection operation may be performed by a number of methods with tournament selection [Banzhaf et al 1998b] and roulette wheel selection [Banzhaf et al 1998c] used widely. With tournament selection, two or more chromosomes are selected randomly and the tournament winner is the individual judged to represent the best solution to the problem – the fittest. The fittest then forms part of the next generation. Typically the tournaments continue until the next generation is fully populated with winners.

In roulette wheel selection, each population member is represented by an individual roulette wheel slot but unlike its Las Vegas counterpart, the slots have varying widths according to the fitness of the chromosome. Selections for the next generation are then made by randomly generating the distance that the simulated roulette wheel ball will travel before landing in a slot and selecting an individual. With a fitter chromosome having a wider slot it is more likely to appear in the next generation. In contrast with tournament selection, roulette wheel does not guarantee to pass on the fittest member of the current population nor will it definitely reject the least fit - though these events are more likely - especially in the early generations before the search has started to converge.

Premature convergence is avoided with the GA technique due to its ability to sample the search space whilst refining the current best solution. This sampling ability is only possible if new parameter / chromosome values are continually created and tried. The initial population, created randomly, contains this necessary diversity though it is quickly lost as the relatively unfit chromosomes are removed and replaced with copies of fitter contestants.

Diversity is reintroduced with the use of the mutation and crossover operators. Mutation strategies are varied but typically involve randomly selecting a chromosome to mutate and then randomly inverting a bit therein. Obviously, non- binary representation mutation must be handled more carefully.

Similarly, crossover operations create new search opportunities by the creation of new chromosomes constructed by exchanging fragments of existing chromosomes.

All these processes have analogies with natural populations: Selection with speciation and extinction; Crossover with reproduction and the sharing and splitting of parent DNA, and Mutation with DNA corruption by oxygen free radicals, radiation etc.

2.6.2 Evolutionary search: Genetic Programming

Genetic Programming (GP) can be seen as a hierarchical extension of GA where the chromosome is replaced by a parse tree built of symbols. The symbols used will reflect the executing environment of the GP and may be statements and constructs in 'C' or VHDL (see 2.7.3.8).

Unlike many other Artificial Intelligence (AI) and Machine Learning (ML) techniques, GP does not need up - front analysis to discover the size of the problem: The relative freedom of the GP trees to grow as necessary in order to define the solution is distinct from, for example, Artificial Neural Networks, where the network structure is usually determined with the aid of heuristics and remains static. With GP, the size (and structure) of the solution produced is part of the answer. Further to GP's ability to self – structure, GP can create and then reuse symbol or program blocks (see 2.7.3.4). This property allows the GP technique to scale up to tackle larger problems if necessary.

2.6.3 GA and GP: Random search?

The extent to which the GA / GP search can be considered random is open to debate, though some random searching will be required to discover unrelated but fruitful areas of the search space, the overall search trajectory will need to be driven. The problem representation used will ideally allow the mutation and crossover operations to largely operate within the immediate neighbourhood of the current search focus but with some more distant sampling. This consideration will allow some hillclimbing to complement the random search action. Beasley [Beasley et al 1993a and 1993b] suggests that GA is a successful technique because it both explores and exploits the search space: The randomness is seen to explore the search space whilst the hillclimbing action can be seen to exploit it.

The operation of a random search and an evolutionary search using GP are directly compared by Haynes [Haynes 1998]. The conclusion drawn here is that random search may be more efficient with low problem complexity but as problems become more complex, and the search area expands, the competitive element of GP becomes necessary to guide the search.

The deterministic nature of all elements of electronic computers implies that any random numbers generated are not truly random: The quality of this process should ensure better sampling of the search space and the option of varied solutions. However, this expectation was not borne out by Meysenburg [Meysenburg & Foster 1999] in their study of randomness and GA performance: In one experiment, the use of an inferior quality random number generator resulted in the best GA performance, and generally no link between generator quality and GA performance could be found. Cantu-Paz [Cantu- Paz 2002] rejects this finding as a general result, and suggests that good generator quality is essential for the initial population at least. Meysenburg has now attempted to explain this anomaly [Meysenburg et al 2002] by suggesting that with poorer generators, the deviation between the idealised and the actual search path taken will be greater with inferior generators; and this can sometimes allow better, though more obscure, solutions to be discovered.

2.6.4 How GA and GP may work

In an attempt to explain the success of GA / GP with some complex problems, two main theories have been offered; Holland's Building Block hypothesis [Holland 1975] and Goldberg's Schema theory [Goldberg 1989].
In the Schema Theory, a schemata is a binary string that may contain either '1', '0' or 'don't care' values so that a given binary chromosome string may be seen to contain a number of different schemata (using the 'don't care' notation). With this view, it can be shown that chromosomes containing schemata having a higher fitness contribution will tend to increase their representation in the population due to the evolutionary action. As a chromosome can contain many schemata substrings simultaneously, it can be seen that there is an implied parallelism in the GA operation, making GA a very efficient computational technique.
The Building Block hypothesis suggests that a GA search will be successful if the schemata are small and relatively independent in terms of their contribution toward overall chromosome fitness. The GA process can then be seen to operate by building upon blocks of high fitness schemata. This hypothesis can be used as a guide for good GA problem encoding, or as a heuristic for the potential success of a GA approach to a problem.
The original schema theory has been criticised because it only relates to the next generation - rather than attempting to predict the overall search action [Whitley 2001][Langdon & Poli 2002].

Although the structure of interacting GP trees may differ, and so complicate any schema theory for GP, exact schema theories that predict schema propagation into the next generation (using single point crossover), have existed for some time [Poli & Langdon 1997]. Poli [Poli 2001] has now produced a specific theory for GP with subtree – swapping crossover and a more general form that may be applied to other types of crossover. In this work the 'don't care' operator may replace a single function or a single terminal.

2.7 Genetic Programming research

A decade after its introduction, GP continues to be a popular research subject; This may be because GP encompasses many aspects of the other metaheuristic search techniques – by result if not by method. Another key factor in the popularity of GP, may be the direct and optimised relationship between the most common symbols of GP – software constructs and statements – and continually improving computer hardware: With a particular microprocessor being optimised to work with a particular language, 'C' for example, any hardware advances will directly carry GP forward. This is especially true in the case of parallel GP (see 2.7.3.7, 2.7.3.9, 2.7.3.10) which can be readily mapped onto the parallel computer hardware that is likely to supersede its ubiquitous serial counterpart.
This direct connection between the target host (embedded computers) and the software, makes GP the practical and efficient choice for the automatic creation of solutions that are both self – structuring, and contain parameters that can be optimised. Thus, the remainder of this chapter is devoted to GP, which is central to this research.

2.7.1 An examination of Genetic Programming research trends

By examination of the Pascal and INSPEC databases hosted at BIDS, the direction of GP research since it first appeared here in 1993, can be traced. The results of this analysis are summarised in Table 2.1.

Year	GP papers	Type A Introducing GP	Type B Improving GP	Type C Applying GP	Type U Unclassified
1993	2	-	-	1	1
1994	5	2	-	1	2
1995	17	6	5	3	3
1996	41	2	9	15	15
1997	97	1	23	48	25
1998	130	13	41	64	12
1999	114	13	37	51	13
2000 & 2001*	724	164	200	158	200
2002*	220	58	63	30	69
2003/1 to 2003/6*	72	30	10	6	26

Table 2.1: GP publications classified by type and year.
[N.B. The original literature source was the BIDS PASCAL database; the latter papers (denoted by *) are available on the BIDS INSPEC database – the PASCAL database being no longer available.]
The papers can generally be sorted into three main categories:

2.7.1.1 Type A Literature: Introducing GP

This literature is concerned with introducing the GP technique as a method for existing problems. The problems addressed have historically been tackled using other methods and GP is suggested as a successful alternative. Though the actual use of GP is typically to model the target system for the purposes of optimisation, classification or prediction, a significant proportion of the content of these papers is often devoted to introducing the basics of GP to the unfamiliar. Generally the work of the 'Introducing GP' literature is to perform a paradigm shift, whereby ideas from GP research are applied to another discipline, rather than progressing the field of GP directly. These papers are usually published in journals whose primary concern is not GP or even computing and typically these involve a basic 'tried – and – tested' implementation of GP. An example of a Type A 'Introducing GP' paper is:

'Animats : Computer-simulated animals in behavioural research' [Watts 1998]

Which was published in the Journal of Animal Science

2.7.1.2 Type B Literature: Improving GP

The thrust of the literature in the B category is toward advancing the GP method rather than applying it to a given problem. These papers are usually published where the primary interest is GP. The focus is toward suggesting and assessing variants on the standard GP mechanism. This includes novel methods of mutation and crossover, alternate ways of managing the GP population, new ways to represent problems, strategies for dealing with operational problems etc. In summary, the Type B 'Improving GP' literature is concerned with enhancing GP operation rather than using it directly as a tool. A good example of a Type B 'Improving GP' paper is:

'Genetic programming with active data selection' [Zhang & Cho 1999]

2.7.1.3 Type C Literature: Applying GP

These papers are concerned with using GP to perform an evolutionary search for an improved result. Some discernible subcategories here include the following:

2.7.1.3.1 Modelling / classification / prediction / regression systems

Here the objective is to evolve a model of the target system. Once created, the model can be used to classify new input or predict output by extrapolation or generalisation from partial or complete input. Example titles are:

'Automated discovery of polynomials by inductive genetic programming' [Nikolaev & Iba 1999]

'A genetic programming approach to rainfall-runoff modelling' [Savic et al 1999]

2.7.1.3.2 Hybrid systems

Usually these combine GP with other Artificial Intelligence technologies such as Artificial Neural Networks to form an improved combined system, for instance:

'Efficient evolution of asymmetric recurrent neural networks' [FigueiraPujol & Poli 1998]

2.7.1.3.3 Co-operative systems

Here the model evolved is to perform optimally in conjunction with other individuals or agents, for example:

'Evolving team Darwin United' [Andre & Teller 1999]

2.7.1.3.4 Applications

- Controllers: usually used to model and so control external, typically motive systems. An example is:

  'Evolving an environment model for robot localisation' [Ebner 1999]

- Data mining systems: the GP system is evolved usually in conjunction with the database, to create more useful access / retrieval systems.

  'A relational data mining tool based on genetic programming' [Martin et al 1998]

- Creative systems: the fitness criteria used for evolutionary assessment here describes music, electronic circuits etc.

'Synthesis of topology and sizing of analogue electrical circuits by means of genetic programming' [Koza et al 2000]

- Program generation: GP is used to evolve programs that meet the required objectives when executed. These programs can also be sequences of hardware blocks rather than software constructs and execute directly in (Field Programmable Gate Array) hardware. An example is:

  'Genetic programming using self-reconfigurable FPGAs' [Sidhu et al 1999]

The Type C 'Applying GP' papers are distinct from the Type A 'Introducing GP' papers (which also involve the application of GP) in that they are targeted at the GP research community directly, have little introductory information on GP and may involve experimentation with a new or improved GP process.

2.7.1.4 Type U Literature: Unclassified papers

The majority of these papers were primarily concerned with computational search methods other than GP i.e. GA or ES; while some were on GP but defied easy classification. Other papers contained the phrase 'genetic programming' used in the context of child psychology, biological genetics or other non-computing disciplines.

2.7.2 GP research trends observed

From this simple examination, it would appear that research into GP may have peaked in years 2000 and 2001: Certainly it would seem that the proportion of type B papers (improving the GP technique) is falling though this may simply indicate that the basic improvements have been investigated. This coincides with a proportional growth in type A papers (introducing GP). Together, these two trends may be indicative of the maturity of GP research; and the consequent spread of this technology into other research fields and applications.

2.7.3 Advances on the basic GP technique

In order to 'frame' this research, and place it in context with other GP research, it is useful to consider other changes to GP:

Since the introduction of GP by Koza nearly a decade ago, the basic technique has been improved in a number of ways, some of these are listed above [2.7.1.2]. In many cases GP is seen as an interesting academic pursuit and targeted at 'toy' problems such as the Santa Fe Ant Trail or the EvenP 5 problem[4].

These efforts at the solution of such 'toy' problems are important because this work may form the substrate upon which the wider adoption of GP is built.

The belief that many Software Engineering problems may be recast as search problems, where GP may be used to search the space of possible programs, is certainly appealing. This attraction increases as the demand for software outstrips the supply of human effort available [1.2].

Improvements upon the basic GP technique continue to be made – new Size Fair Crossover [Langdon 2000] for instance - however GP still needs to scale up to match more complex problems and gain wider application. The scalability is being addressed by the following outlined advances on the original Koza GP. The general areas tackled include:

- Better representation:

  The use of Grammars and Strong Typing can effectively reduce the search space and possibly make it more tractable.

- Problem partitioning and reuse:

  Tackling more complex problems will require more complex solutions. This necessitates GP identifying and using structure within the solution by the evolution of reusable functions and storage of shared work within communal memory.

---

[4] The Santa Fe Ant Trail and EvenP 5 GP benchmarks:
The Santa Fe Ant Trail is a test designed to compare evolved operational strategies. The test comprises a closed environment with 'food' scattered randomly within it, The simulated ant has to move about the environment, collecting all the food, using the allowed motion steps of left, right or ahead; The ant can see food one square ahead. The search motion strategy evolved is run in a loop until all the available food has been recovered. The problem is a deceptive scheduling problem with many local optima.

The EvenP 5 problem requires the evolution of a five bit even parity generator. This test to compare GP methods has been used with many slight variations upon the number of bits, the parity sense, etc.]

- Better implementation:

Finite computing resources demand finite solutions. Dealing with impractical GP tree length or bloat, along with concurrent solutions, will make better use of the resources available for real world application.

- Better fitness assessment:

Finer problems require better targeted solutions. Accuracy, greater reliability and solutions that are generally a better and more concise match to the problem are needed. The use of (possibly) interactive tools to guide the solution of problems with a aesthetic dimension, may prove to be a most fruitful research area.

These issues are widely inter-related and addressed in some proportion by the technical advances detailed below.

2.7.3.1 Strongly typed GP

Allowing a GP system to perform arbitrary crossover and mutation operations can easily lead to the creation of both semantically and syntactically invalid code. For instance, a terminal value that acts as a parameter for a function call may need to be of a given type. Another example would be an operator that requires two arguments might only be provided with one, subsequent to code mutation or crossover. The code may be described as 'closed' if such problems are avoided. Much early research into GP, including some of Koza's early work, circumvented this problem by simply evolving in the LISP language.

The semantic correctness of evolved code is of potential benefit if the GP system is being used to discover new aspects of the problem domain, or if some confidence is required in the validity of the relationships described. In these situations, the explanatory power afforded by semantically correct 'readable' code is desirable. Alternatively, one of the motivations for using GP is to allow its inherent creativity to attempt to produce better solutions than human programmers. This 'black box' approach would generally be restricted by the imposition of demands for readable code.

Syntactic correctness however, is more clear cut; If the code being evolved by our GP system is to be compiled or interpreted before being executed, then the evolution process will simply fail because the code will not run. However, if the code symbols used are not those for a validated compiler (such as the Ada compiler) but are lines of Assembler - level constructs [Nordin et al 1999] or 'C' [O'Neill & Ryan 1999] then the code may exploit undocumented and unsupported features of the hardware or software environment. This code may then operate in ways that are not repeatable or transportable for use beyond the immediate context. In the case of GP code that describes hardware, such unrestrained evolution may well lead to the destruction of the host hardware (see 2.7.3.8).

The use of strongly typed GP allows evolution to operate between the above two extremes i.e. neither too constrained so as to restrict creativity unduly nor too wild so as to be of restricted use. Strong typing can be seen to restrict the search space usefully and in its simplest form the types available at mutation and creation are restricted.

Alternatives to strong typing include the use of Context Free Grammars [Whigham 1995]. discussed in section 2.7.3.6 and the use of polymorphism. Martin [Martin 2000] uses a polymorphic approach in a prototype system to evolve intelligent telecommunications networks. With his technique, values of different types are maintained simultaneously in memory and chosen accordingly. A similar approach is to coerce the stored variables at run time to the appropriate type. As Martin points out, simple coercion can often be meaningless; for example, the coercion of a Boolean type to a telephone number - in addition to being computationally expensive.

2.7.3.2 Program bloat

The unrestrained action of crossover in GP can lead to excessive program length when already lengthy trees are combined. It is possible that much of the code in these trees is never executed or, if executed, adds nothing to the task at hand. Such redundant code sections have become known as 'introns', and these can obviously impact the real world use of GP for practical applications due to their requirement for storage and possibly processing resources. It is clear that the presence of introns during evolution (or at least the initial stages) should be tolerated as the carriage of unused genetic material will add to diversity. This stored diversity can lead to improved sampling, when sections of this dormant code material are called into use at a later stage by the actions of crossover or mutation. The growth of introns can also be favoured by the evolutionary process because these code portions form 'padding' to protect active and successful code sections from destruction by crossover or mutation. Unfortunately, with computer resources being finite, code bloat in GP solutions needs to be addressed. A simple solution is to add tree length to the fitness assessment criteria and hence evolve out excessive solutions during the closing stages of evolution. Simple length alone however, may not be a very useful measure of usefulness: Podgorelec [Podgorelec & Kokol 2000] states that 'software is the most complex of human constructions' and attempts to use a fractal complexity measure to attack program bloat. With this method, Podgorelec uses a correlation measure to distinguish between random and useful sections of code. The resulting fractal structure overview of the code allows for more intelligent pruning of the GP trees.

2.7.3.3 Tuning GP

As with many other AI techniques, tuning or optimisation can be associated with overfitting the training set and typically occurs at the expense of generalizational abilities of the trained system. Yeun [Yeun et al 2000] attempts to tune his GP system and avoid such problems by using a two stage approach: Firstly the primary GP trees are evolved in the usual way by application of a training set and assessing tree fitness in the current generation. These primary trees will form a possibly sub – optimal function approximation to the target solution. The next stage is to form auxiliary trees using a Linear Associated Memory technique that assigns weights to nodes in the primary tree. These weights are then selected by adjusting their values by a small random factor and applying a portion of the training set. The overall effect allows the system to creep towards a global optimum.

A different two stage approach was adopted by Howard [Howard et al 1999] to create a system to detect ocean targets from radar images. Here a first stage GP detector is trained using clear ocean and perfect target images. Next this detector is fed real images and the false positives recorded. A second detector is then trained using the false positives data and the clear images. Finally a composite detector is created by using the output of the first detector to qualify the output of the second. Howard also suggests a technique whereby the constants in a GP tree may be optimised between generations using a GA approach.

2.7.3.4 Function evolution

In order for GP to be a tool for use against larger practical problems it should be scalable. One method of improving scalability is to ensure that the GP solution can be expanded to match the problem. Ideally such an expansion would need to be made in such a way that the evolutionary process undertaken was still a practical proposition. One way of achieving this goal is to encourage the development of some form of hierarchical structure of re - used code units. Koza [Koza 1992c] describes such items as Automatically Defined Functions and these include subroutines, functions, loops and iteration. The insertion of such reusable code units effectively allow the problem to be partitioned in a useful way.

There are several basic operations relevant when ADF subroutines (for example) are required; Subroutine duplication, creation and deletion, and parameter duplication and deletion. The process of subroutine creation involves choosing a point in a GP tree at random and working downward to a second random point. The complete code section is removed and stored separately as a subroutine and a reference to it replaces it in the tree body.

In the duplication operation, an existing subroutine in the subroutine store is copied and references to the original subroutine are replaced randomly by references to the duplicate. The effects of mutation and crossover will then cause the duplicate and the original to diverge.

A slightly different approach to GP with ADFs is described by Ahiuwalia [Ahluwalia & Bull 1998]. In this system, a dynamic library of functions is maintained. The functions are formed in a similar way by compressing sections of the GP tree and transferring them to a function library. Diversity is maintained by the occasional expansion of a library tree back into the GP tree population. The approach here is to randomly swap calls to the library functions as the trees are evolved. Functions that are infrequently used as the generations progress are dropped from the library. Ahiuwalia's system is intended to progress both the functions in the dynamic library and the GP trees by preventing a less fit function 'hitching a ride' on a fit free or vice – versa.

2.7.3.5 Using memory

The learning, storage and re – use of information is a method widely used to enable higher forms of life to tackle more complex problems. In GP, Koza [Koza 1992d] suggests many ways in which knowledge, instilled over generations, can be stored in memory added to the system. These include more complex forms such as indexed memory with two degrees of access. Expanding on this, Haynes {Haynes 1998] uses a collective memory system where the search task is split by the use of collective memory to store partial solutions. The collective memory or knowledge base here is filled by evolved search agents with direct access to the search space. In turn, these stored partial solutions are used by evolved process agents to complete the system. Forcing a separation of the search space from the secondary process agents in this way forces the evolution of a system that can partition the problem successfully.

2.7.3.6 Grammars

Grammars have been used to abstract the problem away from the space of program symbols and constructs. This can be seen to make difficult problems more straightforward and Leung [Leung & Wong 1995] uses a Logic Grammar representation to outperform a system utilising Koza ADFs.
Ryan [Ryan & Ivan 2000] has created a system called Grammatical Evolution (GE) whereby the solution symbols are represented in binary in a variable length chromosome, similar to that used in a GA. Mapping from the phenotype to the genotype is performed using Bachus Naur Form rules. This ensures that the program code ultimately produced will be valid, and the target language variable. The GE system is also shown to deal with the problem of program bloat.

2.7.3.7 Parallel GP populations

The advantages afforded by the possibility of simultaneous exploration of the search space in GP, can be compounded by the use of multi–processor platforms to perform fast concurrent evolution or execution of the GP. Typically parallel GP systems divide the total population into a number of subpopulations or demes and these demes evolve largely in isolation.

Diversity (and hence new search trajectories) are facilitated by exchanging some of the least fit members of one subpopulation with fit members from another. Fernandez [Fernandez et al 2000] combines such an island model with a client - server model to manage the exchange of members between demes. In this paper, Fernandez establishes some heuristics about population size, number of populations, exchange strategies and rates etc, Of course, these parameter choices needed for parallel GP are additional to those facing the designer of a standard GP system.

2.7.3.8 Hardware concurrent GP

Field Programmable Gate Arrays (FPGAs) are dynamically reconfigurable integrated circuits (ICs) that contain arrays of generic hardware blocks and interconnection / gating logic. The function of these blocks and their interconnection and gating is determined by the bit pattern held in the underlying configuration memory. This bit pattern, and hence the overall FPGA function, can be downloaded to the device from a computer or from another source.
The FPGA has two major application advantages: Firstly it can be used to integrate and replace a large amount of discrete ICs thus making compact, low cost and low power electronic hardware possible. Secondly, the FPGA allows for easy to maintain or dynamic solutions in much the same way that embedded software has the advantage over hardware in engineering solutions. Both these attributes make the FPGA of interest to GP research and typically the FPGA is used to execute the GP using hardware function blocks as the symbols rather than the more usual software constructs and operations of a software – implemented GP.
Executing and evolving the GP in hardware in this way, has many advantages [Koza 1992e] that mainly stem from the natural concurrency that can enable parallel execution and pipelining of the GP trees during both execution and evolution stages. In addition, this parallel execution can operate at up to hardware gating speed.
Reetinder [Reetinder et al 1999] suggests the use of the FPGA as a self – contained evolutionary black box whereby an initially random configuration bit pattern is loaded into the FPGA. The fitness of the hardware solution defined by the bit pattern is assessed according to the ability of the configured device to perform the target task. This approach, though computationally efficient,

has some drawbacks in that it can evolve designs that use artefacts of that particular device and its current environment that are not transportable; For example, ambient temperature and humidity that can affect the execution speed of the device or the reliance on parasitic or undocumented (and so inconsistent) artefacts of the device. Indeed, in some FPGA devices an arbitrary configuration pattern can destroy the device because active outputs may be connected together.

Popp [Popp et al 1998] uses a hybrid approach whereby the ultimate GP code is executed in the FPGA hardware but the evolution occurs (with useful constraints) elsewhere. Popp uses Very high speed integrated circuit Hardware Definition Language (VHDL) rather than normal software to form an algorithmic description of the solution in the GP parse trees.

This VHDL description and definition of the FPGA function is simulated to assess its fitness externally. Finally the evolved VHDL solution is compiled and downloaded to be executed in hardware. In addition to the VHDL description giving a degree of explanatory power to the solution, the time consuming steps of post - compilation design placement, routing and downloading are avoided. However, the use of a simulation to model the FPGA operation in abstraction should be noted as a disadvantage of this method.

A different attempt at GP on parallel hardware was made by Poli [Poli 1999] where the bitwise parallel processing that occurs in serial computers is exploited. In this paper, Poli describes how the 32 bit positions of a 32 bit wide machine can be viewed as 32 separate 1 bit processors operating concurrently on the same instruction. This view is restricted to operations whereby register bit – neighbors operate completely independently i.e. bitwise logical instructions. This novel form of Single Instruction Multiple Data parallel processing was shown to improve the processing speed of GP type operations by 1.5 to 1.8 orders of magnitude under certain circumstances.

2.7.3.9 Slicing / Parallelized concurrent GP

As an alternative to evolving programs explicitly for parallel hardware, existing code may be parallelized: This approach will enable the performance leaps offered by parallel platforms to be utilised by legacy software that was created for serial computation. The re – engineering of old, poorly maintained and possibly undocumented code is an onerous task - without the added burden that parallelization presents. Previously, the methods employed in parallelization included manual translation or the use of parallelizing compilers. The main work involves the decomposition and distribution of the processing involved in program loops amongst the processors available. The secondary task of distributing the sequential code fragments is less problematic. A successful parallelization will not only share the burden of iteration but also ensure that this sharing scheme minimises any waiting necessitated by program loop dependencies. Williams [Williams & Williams 1999] describes four basic loop conversions that may be tried: Loop fusion - where loops are amalgamated; Loop splicing - where a single loop is expanded into two; Loop interchange - where the inner and outer loops of a loop nest are exchanged, and finally, Loop reversal - where the order of loop execution is reversed.

Williams uses a GP approach to attack this complex problem with two strategies; Gene Transformation and Gene Statement. With Gene Transformation, the program loops are numbered and a useful list of sequential loop transformations evolved. The focus of Gene Statement is on the code constructs themselves.

Ostrowski [Ostrowski & Reynolds 1998] introduces program slicing as a method of program decomposition where a slice is an equivalent program extract that tracks all the program statements that effect a given variable for a given input range.

Combining both program slicing and automatic parallelization, the Paragen system [Ryan & Ivan 2000] will slice a program in order that GP equivalents to the slice can be evolved. These equivalents are then readily ordered for distribution across a parallel array. Ultimately the Paragen system may become a powerful tool that will re – engineer (and maintain) existing programs into an equivalent version in any language and for any platform.

2.7.3.10 Task sharing GP systems

The use of multiple agents or subprograms that may be evolved to collectively solve complex problems efficiently has been researched in various formats: Angeline [Angeline 1997] created a system of Multiple Interacting Programs in which number of parse trees are evolved that can share intermediate variables. Each tree is associated with one output and intermediate variable pair, and is evolved in isolation. This can be seen as a method of partitioning a problem and so scale the problem into a manageable form. This automatic problem partitioning effect is similar to that of Haynes mentioned earlier (see 2.7.3.5). Two distinct types of problem decomposition are investigated by Vallejo [Vallejo & Ramos 2000] whilst attempting to evolve an insect locomotion simulation. In the first experiment, Task Sharing between six subprograms (one per leg) is evolved - with the legs operating independently - but having access to a common binary memory containing current leg status data. In the second experiment, the more successful method of Result Sharing was investigated and in this case the evolutionary fitness assessment incorporated a measure of how well the subprograms worked together in co-ordinating the legs.

2.7.3.11 N version GP

The use of concurrent voting systems or teams, using N different versions (N Version Programming or NVP) to enhance the reliability of software systems has been the subject of research using both human and evolved GP software.

2.7.3.11.1 Human NVP experience research

Hatton [Hatton 1997] investigated human NVP and clearly demonstrated the reliability benefits obtainable with this method by comparing the rate of faults between the best in a population of versions, and a voting team drawn from that population. The use of this method in software self - diagnosis is also highlighted here. Clearly the advantages of having N versions voting on a decision is only of value (in terms of software fault tolerance) if those versions differ usefully. Homogeneous voting teams may have value in situations where hardware fault tolerance is an issue.
With the focus on software fault susceptibility, Lyu [Lyu 1993] suggests four ways in which software diverges:

(1) Diversity in the software structure.

(2) Diversity in the fault quantity within the software.

(3) Tough - spot diversity which measures the fault quantity in 'awkward' areas of the software; And lastly.

(4) Differences in the way in which the software fails.

Lyu's paper also suggests methods for forcing diversity across the design teams by the use of different languages, design methodologies etc.

2.7.3.11.2 GP NVP attempts research

The use of evolved GP teams to create an N version voting system is an attractive idea because of the cost of human development effort (which will continue with the need for N version maintenance) and the difficulty in achieving design diversity with humans. With an evolutionary approach, two of Lyu's four diversity types can be directly related to the stages of genotype and phenotype creation; Genotype diversity is related to program structure whilst phenotype diversity is indicated by the failure behaviour of the system itself. Feldt [Feldt 1998] identifies the three classes of GP genotype diversity that can be manipulated as:

(1) Those concerned with the search space (functions, terminals, any forced program structure such as ADFs).

(2) Those that are concerned with the search (population size, mutation rate etc.).

(3) Those to do with the evolution itself (number of test cases etc.).

Bongard [Bongard 2000] provides a heterogeneity (or social entropy here) metric that is used in his co-operative GP Legion system: Unfortunately, the measure is directly dependent on the fitness measure used in evolution, that is, problem specific.
Further GP N version research by Soule [Soule 1999] considers how the voting balance can be used to give useful feedback on how efficiently the system has tackled the problem: For instance, a large majority vote may indicate a waste of resources such as that caused by poor input partitioning. Similarly, the voting scheme can be seen to introduce a form of hysteresis into the system acting to evolve more robust solutions than a single channel GP solution - though utilising more resources.

2.8 Future mass search paradigms

The use of parallel hardware combined with the decomposition of problems, may be one answer to certain hard computational search problems; two promising alternatives that may allow a 'brute force' attack (whereby all possible solutions are tested) have been suggested: Both of these techniques have the potential to circumvent the upper limit on data processing calculated by Bremermann for classical computers [Bremermann 1962] of 2x10^47 bits per second per gram of its mass.

2.8.1 DNA computing

Here candidate solutions can be encoded at a molecular level in DNA and the solutions tested using biochemistry. The testing stage may take hours but the payback lies in the mass parallelism afforded by the molecular encoding; it is suggested that 10^23 operations maybe encoded in just two test tubes of DNA. This would seem to make hard problems such as message decryption possible where an unknown 56 bit key (e.g. the DES encryption standard) was used.

A recent paper attempts to bring DNA computing slightly nearer by using the DNA material to construct nano-scale computing elements: Pelletier & Weinerskirch [Pelletier & Weinerskirch 2002] describe a process for the construction of three dimensional computing arrays consisting of bit multipliers and other elements, that could offer true mass computing.

2.8.2 Quantum computing

According to Spector [Spector 2002]. quantum computers would be able to perform exponentially many parts of the same calculation simultaneously: Recent progress on the path to building one of these astonishing (though currently hypothetical) machines, has been the reliable sensing of the quantum computing element (Qubit) state - without changing that state. Spector [Spector et al 1998] has used GP to evolve algorithms that would give better-than-classical computer performance - when ultimately run on quantum computers - by simulating the operation of Qubits. Spector suggests that the quantum parallelism and inherent probabilistic processing of these machines should make them ideal for searching program space by GP, yielding better-than-classic evolution speedups.

2.9 Chapter summary

In this chapter, the main forms of computational search have been described, and the use of GP has been detailed: By examining the body of published GP literature, three general types of research have been identified, namely; Introducing GP, Improving GP and Applying GP. As the main thrust of this work involves the application of GP, this chapter has focussed on changes in the original GP technique that have resulted in improvements in its effectiveness, efficiency and applicability. Lastly, two new computing paradigms have been described that promise to exponentially expand the power of computing – far beyond its current steady growth – further facilitating search.

This examination of published GP research highlights the lack of work involving the use of GP for the creation of application programs for real problems: The aim of this research is to address this omission by suggesting a software creation method targeted at embedded control applications – which is the subject of the next chapter.

3 Control systems

3.1 Introduction

This chapter characterises embedded control systems, explains why they are a suitable problem domain and then outlines an approach that can be used for the automatic creation of software for certain control applications.

3.2 Embedded control systems: Dedicated computing

An embedded control system will typically comprise of dedicated hardware hosting dedicated embedded software or 'firmware' (i.e. between 'hardware' and 'software'). This is distinct from the ubiquitous PC situation, which is a more familiar small – scale computing operation to most people. With the PC, the software applications executed share the machine with the Operating System (OS) and other applications; each application will appear to temporarily convert the general – purpose PC hardware toward a specific function i.e. to turn the whole into a word processor, for instance.

In the case of embedded software, it is usually expected that the hardware / software will perform one application only: Examples of such applications can include mobile 'phones, microwave ovens or network switches. Usually the rest of the controlled equipment is so targeted at one function, that the total dedication of the control system to that function can be advantageous: The consequent removal of any requirement for design flexibility for either hardware or software, allows for the paring – down of most ancillary functionality. In the case of the controller hardware, this will allow the designers to usually choose hardware that will perform exactly what is needed and no more. This consideration leads to hardware with minimal size, weight, power consumption, operating speed and cost. This ability to minimise and tailor hardware is particularly important with mass – produced items such as video recorders, MP – 3 players etc. An existing Evolutionary Computing (EC) example of such minimisation for mass production uses GA search to discover the longest execution path (time) in a car engine management computer. Knowledge of the worst case cycle time will enable the designer to choose a minimal, lower performance (lower cost) processor for the management role with confidence [Gross 2000].

The workhorse of embedded control is the microprocessor and frequently a version of this device aimed specifically at embedded control - the microcontroller – can be used. A microcontroller has a microprocessor core but with many of the other hardware functions (more usually performed by other hardware 'support' devices around the microprocessor) integrated onto the microcontroller chip. In extreme cases, the address and data busses of the microprocessor do not leave the device due to on – chip program and data memory etc. This will free the pins of the microcontroller chip for relatively direct connection to the rest of the equipment i.e. lights, switches, communication links etc., further minimising the design.
Similarly, the dedication of the control system to one application will allow for the use of minimal software that only has to perform the task at hand – often to the exclusion of an OS (because there is no hardware sharing or porting) allowing the application software to talk to the hardware directly. This cut - down software requirement will further reduce speed, cost, size and other design demands.

Literature devoted to embedded application development appears to be scarce; for further consideration of real time, low level embedded programming [Leventhal 1978] can be consulted: Alternatively, the hardware and software issues relating to microprocessor – level interfacing are examined in [Zaks & Lesea 1979].

3.2.1 Comparison between embedded control and other applications

The following table [Table 3.1] attempts to highlight areas where embedded control applications generally differ from those of other computer applications:

	Embedded control Application	General computer Application
Data interface to world	Varied & complex: may require conversion	GUI, printer, keyboard, network etc
Platform	Optimised, specific	Generic
Platform cost	Critical, minimised	Less important
Response timing	Critical	Less important
Aesthetics	Minor issue	More important
Pressure to upgrade	Little	Competition
Development expertise	Essential	Not always

Table 3.1: Comparison between embedded control and other computer applications.

3.3 Temporal and operational hierarchies in embedded control systems

Another distinct feature of embedded control systems relates to their 'real – time' nature: The majority of embedded control systems operate continuously and immediately – often to meet strict response time criteria. This is both possible (due to the dedication discussed above) and essential due to the nature of some control applications. For example, the response time of a web browser is less of an issue when compared to the response time for an emergency stop input on a hydraulic ram controller.

Prioritised processing is achieved – and can be guaranteed – in embedded control systems by using exception or interrupt processing. Typically, this is a hardware facility that will force the processor to run specific pieces of code in response to certain events. These events could be the activation of a sensor, activity on a communications link or the expiration of a timer. Generally, necessary execution priorities can be set up when the computing system is designed: This will allow events such as data storage and safe shutdown (in response to a power failure signal) to take precedence over keyboard polling, for instance. Mixed in with these, timer interrupts can be used to sample an analogue input or to update the position error in a motion control system: Multitasking OS simulate multiprocessor operation and resource sharing on a single – processor platform with the use of context switching initiated under such timed exception processing.

3.4 Multiple inputs and multiple outputs

A further distinct feature common to many embedded control systems is the presence of multiple inputs and outputs: A building management control system, for instance, would need inputs for temperature, humidity, time-of-day etc; and output demand values for temperature and humidity, control lights and door locks etc: A missile would need to input its relative position in space in three dimensions, and produce three separate outputs to correct that position.

These inputs and outputs are all distinct hardware ports and consequently distinct data sources and sinks. Again, the necessity that embedded control systems need to integrate with the multi – dimensional real world forces this structure on the hardware and, consequently, on the software.

3.5 Task decomposition in embedded control systems

Exception processing on dedicated platforms is used as a method of guaranteeing response times and ordering processing priority, whilst the requirement for multiple input and output streams further segments the code. Indirectly, these software structure frameworks automatically deal with some of the complexity inherent in these applications by creating operational 'channels' linking inputs to outputs across execution space, and through processing time. The code in such channels may be then treated in isolation to some degree, and hence the code for the overall application may be created or examined in a more piecemeal fashion.

The idea of examining software by starting at the outputs and working backwards down the 'channel' to discover the causal relationships that gave rise to that output, has already been the subject of test and maintenance research [2.7.3.9].

3.6 Using application decomposition: Automatic creation of control software.

The idea behind this research is that by targeting applications of this type – with the inherent structuring and decomposition of the software, the threads of code used to form the channels (described above) can be, for a certain class of application, simpler; simple enough to create automatically.

Existing forms of genetic programming (GP) create software automatically but an examination of GP literature [2.7] reveals few, real instances of code evolved to perform a specific function (of any real complexity) for real world application. Perhaps this is not surprising given that creating computer programs by GP can be seen as a Markov process[5] and, as such, the longer the program is (and so the greater the linear complexity) then the probability of its discovery will decrease geometrically.

The aim of this work then, is to suggest a method of ameliorating the program discovery task by suitable decomposition until the problem becomes tractable by evolutionary search. Even then, the pragmatic view will be to restrict scope of this work to a subset of software design problems which are, already, relatively simple.

3.7 A suitable class of problems

The idea of using automated methods for generating control software inevitably leads to issues of trust being raised: Accuracy, reliability and robustness may all appear compromised due to the lack of human intervention – even though human design is demonstrably far from fault-free.

This research suggests an alternative method for software creation but realistically this will only be applicable to a subset of software problems. This subset is a particular class of problems that can be loosely defined by identifying;

(1) The type of problems that are wholly unsuitable

(2) The type of problem which is more appropriate:

---

[5] Markov processes:
The construction of an LGP string can be seen as a Markov process:
A Markov process is a discrete random process whereby the probability of moving onto another state depends solely on the current state of the process and not on any history or other stored information. Thus the choice of the next statement in the construction of a GP tree / string can be viewed as an independent and undirected event for that individual. This model has been used to construct a analysis of GP operation [Langdon & Poli 2002].

3.7.1 Prohibitive problem features

- Safety – critical problems

The ability for an evolved program to find a perfectly fit, though incorrect, solution resulting from an ambiguous fitness function is very possible: Typically, this can occur because the fitness function used is an abstraction of the real problem. Such invalid solutions may well be transparent to automatic testing generated in the same manner.
Even with the adoption of measures to prevent ambiguous solutions being used, the notion of using machine – designed software in situations where lives could be lost would be the act of a brave – or misinformed - vendor.
Such considerations about the real and perceived trust in a product would easily prohibit this software creation method being used in applications such as lift controllers, missiles, aviation systems etc.

- Problems requiring solutions of high complexity

Problems that could not be decomposed and would always require an evolved solution with even a relatively low level of computational complexity, would be problematic - simply because the search space would be vast, and the probability of discovering the solution consequently small.
Applications such as signal processing or compilation can be included in this category.

- Problems requiring high reliability or robustness

The advantage in creating software automatically is that there is no human input; in this act, the actual nature of the code being run can be unknown, and as a consequence, the potential for unreliable operation exists as real – world execution uncovers unforeseen problems: Aside from the safety – critical issue, the simple problems of trying to sell an unreliable product will preclude an automatic approach for many applications. Examples here could include assembly – line control / monitoring applications or access / entry control applications.

- Problems requiring high accuracy

Problems requiring solutions than can offer high precision output will typically present a higher search burden because the evolved solution will need to be almost exact. Applications such as accounting packages and computer – controlled tools are examples of such inappropriate targets.

- Problems requiring defined response timing

Demanding guaranteed response timing, or fixed execution time in general, would necessitate the evolution of temporal execution hierarchies on top of the structural processing hierarchy and present an unreasonable burden upon the system proposed here. This could exclude targets such as high speed communication systems, games programs or the safe control of moving objects.

- Problems with complex inputs or outputs

The necessity to define fitness functions that will accurately allow the EC to find the correct solution would be immensely difficult for situations where complex input or output had to be handled: Consider the case of ASCII keyboard input or GUI output. In cases where the output has a subjective or aesthetic dimension, it may never be possible to express the required solution explicitly to a computer: Indeed, the effort involved in capturing such a fitness expression could easily outweigh any labour – saving advantage promised by automatic methods.

3.7.2 Suitable problem features

Beyond the applications that do not fall into the above categories of unsuitable problems, the following outline problem characteristics that can facilitate the proposed technique:

- Problems that can be decomposed appropriately

These are problems where the solution may be, or has to be, realised with the use of multiple outputs and where the code producing that output can be formed as a channel fed from one or more inputs. The ability to segment a larger and more complex problem into a number of simpler code channels is central to the 'divide and conquer' strategy of this work.

- Mundane problems

These are problems whose solution may not be perceived as 'exciting' or 'challenging' work, making the staffing of such projects an issue [1.2].

- Support for product designers

This system could be used to produce code for product designers who wish to use embedded control technology but who do not have the time, interest or budget to become embedded control programmers.

Consideration of the above criteria, and those prohibitive criteria outlined in [3.7.1], classify target problems as ideally being non safety – critical: Simple (when decomposed) problems that do not require precision, undue reliability or defined response times.

A subset of embedded control applications can be seen to fit this description. Thus examples of such applications could include fridges, TV remote controls, building environment management systems and dishwashers.

3.8 How application decomposition can be used

In the embedded control systems categorised at the start of this chapter, the problem decomposition is an artefact of the necessity to prioritise and divide execution by using both ideas of processing order and output requirements. To segment arbitrary control applications along these lines would not be practical automatically because this would require an unreasonable amount of input from the product designer: Instead, the approach is to segment the application by output alone - because the product output requirements would be known to the target system user (the product designer) [1.4]. The decomposition of control problems by output alone (and not execution priority) will inevitably preclude problems where guaranteed response times are essential; but this restriction upon the target problem class is viewed as an acceptable compromise [3.7.2].

With this segmentation approach, each output will be the evolved output of (at least) one execution channel that has visibility of all system inputs during training – to use as necessary. The channel code is evolved, and later run, as an isolated concurrent processing task (thread). Synchronisation and cycling of the tasks can be organised by a pre – written multi – tasking OS. The OS will have the responsibility of cycling the channel code continuously, collect the task output data and update the system outputs. To an extent, the technique is (concurrently) scalable in that the evolved system may be expanded simply by adding further tasks and outputs.

3.9 Problem capture; EC training set generation

As previously discussed [1.4] the product designer would be aware of the inputs and outputs necessary to interface the control computer to the rest of the product. This knowledge then, would have to be captured in order to automatically create the control system: Though beyond the scope of this research, such a 'front – end' tool would need to be constructed to question the product designer upon output and input quantity, type and range. Additionally, the relationships and inter-relationships between these interfaces would need to be captured; This could be achieved with the use of formula input, truth tables, relationship graphs or graphically, .

The output of this front end tool would be a sufficient training and test set relating the system inputs and outputs, interface types, number of tasks and initial EC parameters, necessary for the EC system proposed here to create the software.

3.10 Possible generic hardware host

This chapter suggests a system for the automatic creation of concurrent control programs; the next chapter details how a demonstration system was implemented for two simulated applications. For this work, the concurrent processing is achieved by using a multi – tasking OS: Ultimately, each task could be run on a dedicated single processor in a multi – processor hardware implementation. Further overall design optimisation could be afforded by hosting the evolved code in low – cost, mass – produced, generic, multi – processor microcontroller blocks. In addition to a pre – written OS, these blocks could come supplied with network interfaces for the download of the evolved code (invisible maintenance), redesign of the hardware or even the provision of an uplink for monitoring purposes.

3.11 Chapter summary

This chapter has outlined the characteristics that make typical embedded control systems distinct from other software applications; and the suitability of such architectures to automatic software creation by EC, have been suggested.
The next chapter describes the construction of a demonstration system to simulate the operation of the automatic software creation scheme proposed here: In addition, two suitable problems are described, and the latter, larger problem of a Washing Machine controller is tackled.

4. The experimental system

4.1 Introduction

This chapter describes the experimental system used to demonstrate the automatic creation of a control system, as outlined in the previous chapter: This will include details of the Evolutionary Computing (EC) approach used, the evolutionary language set and data types, the simulated Operating System (OS) operation and the software tools used. Finally, the two demonstration applications - the fridge and the washing machine - are described: This latter problem (being effectively a superset of the first) is the subject of the next chapter, in which the experimental demonstration system is built and adapted.

4.2 EC technique used

The choice of EC technique falls naturally to Genetic Programming (GP): In addition to being a consistent and major research subject, the ultimate objective here is to create solutions (programs) that could be embedded, and so run directly on the target hardware.

Standard GP will produce tree-like structures that can easily represent the parse tree of the program being generated. These trees have the ability to depict the complex structures possible in many programming languages. However, trees also create some difficulties; arguably the most detrimental (especially if the target is a real – world implementation) is runaway program size or 'bloat' [2.7.3.2].

A pragmatic choice for this work was to use a form of Linear GP [Banzhaf et al 1998a] which replaces a tree-like structure with a variable-length linear string representation; typically not containing any form of program branching i.e. the execution flow is linear. Straight-away the problems of excessive code growth reduce because the code can only expand in one dimension up to a limit: Arbitrarily large trees are not connected together during crossover, or attached during mutation.

Though a linear representation is not familiar to human programmers, it is arguable that linear representations, are in many senses, equivalent to tree representations if they span the same program space once decoded [Albuquerque 2000]. Thus LGP can perhaps be seen as a useful compromise between the technique (EC) and the media (computer software).

4.3 Evolutionary language set

Classic (tree) GP would typically allow a (near) full implementation of the chosen programming language: For this work, the language set available for evolution is a restricted subset of the functions that are available in a basic programming language like 'C' or BASIC.
This move away from a full implementation of the chosen programming language is seen as an acceptable and possibly advantageous step: All the useful possibilities of the language remain available – albeit in a more verbose (and perhaps safer) form. As can be seen in appendix A and [4.3.2] below, the grammar facilitates all useful continuous mathematical and logical functions.
This use of basic 'general – purpose' program operations also reflects the desire to create a system that can create programs for a variety of targets; which all fall within the problem class identified - i.e. the ultimate applications may range from balloon inflation to bird feed. Alternatively, if the target class was known and static, then a controller for an automatic lawn sprinkler might, more usefully, use trigonometric functions as language set primitives.
The language set here, is a slight departure from 'standard' LGP in that the program flow is not strictly linear: A conditional branch instruction is included to allow forward branching upon the TRUE / FALSE interpretation of the IF condition case. The intention here is to allow for the evolution of equivalents for the standard forms of iteration found in computer programs; gotos, subroutines, branches and loops.
This simplification of the instruction set is similar to the approach taken in Reduced Instruction Set Computing (RISC)[6] and this may appear to be a regressive step in software engineering terms, that can frequently result in the creation of unreadable code, However, recall that the programs are to be generated automatically and it is not the intention that humans would need to examine the code. A RISC approach of simple program statements, will allow a near one to one transformation to machine code which offers the benefits of

[6] Reduced Instruction Set Computing, RISC:
RISC processors use what is perhaps a counter – intuitive approach whereby the machine-level instruction set is limited to simple instructions (Add, OR etc.) and more complex instructions that will perform complex, compound operations are omitted. Generally, this will result in longer machine code programs after compilation because more instructions will be required to accomplish the same task; however, the hardware design of the processor chip is now simpler. This simplification will allow the processor to be run faster such that even with a greater number of instructions to process, the overall execution time will usually be reduced. [For further information: Stallings 1990]

speed / compactness and portability (since almost all processors have similar basic machine level operations).

4.3.1 Program string statements

The program representation used here consists of a variable length string of words (up to a practical maximum which will depend on the computing resources available) where each word represents a single, atomic statement. With the exception of the IF branch instruction, each statement will have one or two inputs and an output term: An input source maybe connected to an input terminal, a numbered memory, a constant or a subtotal accumulator. The output term maybe connected to a program output terminal, a numbered memory or to the subtotal accumulator register. The subtotal accumulator was included to facilitate the flow and construction of information up through the program by a path that could be readily used (easily found during evolution). However, the subtotal accumulator can be seen as simply another (though directly – addressable) memory.

4.3.2 Program functions available

The available functions that will determine the operation of each program statement are as follows:

Mathematical operators:

+	(Add)
-	(Subtract)
*	(Multiply)
/	(Divide – protected by returning 1 upon attempts to divide by zero)

Logical operators:

AND
OR
XOR
NOT

< (Less than)

> (Greater than)

Branch:

IF [condition] (jump forward to) Statement number

4.3.3 Suitability of language set

The general - purpose (though computationally efficient) nature of the chosen language set [4.3.2 above] may be seen by examining the more specific language set choices made by other researchers:
In the production of an Automatic Theorem Prover, Nordin [Nordin et al 1999] used a function set of inference rules (such as 'X+0 may be replaced with X') to evolve proofs for mathematical functions; Popp [Popp 1998] uses VHDL[7] primitives to evolve Field Programmable Gate Array (FPGA) – hosted hardware designs; and Martin [Martin 2000] uses functions with two levels of abstraction (e.g. 'FROUTE' to evaluate routes and 'SendMSG' to deliver a message) for automatic network service creation using GP.
These are all examples of very specific language sets, targeted at particular problems: This research proposes a general – purpose system for evolving solutions for any control problem from the identified class [3.7], hence a general, low – level language set with non (application) specific instructions was used.

4.4 Data types available

During evolution, the system can use two data types: continuous and logical. Continuous data type can be positive or negative floating point numbers; the logical data type will only be either '1' or '0'.
The values are (at this time) both held in the same storage and the type to which the stored values are applied is interpreted in context to the instruction at hand: For instance, the logical instructions above will interpret all arguments as unclamped logical values – so that zero is interpreted as FALSE,

[7] VHDL:
Very high speed Integrated circuit Hardware Description Language is used to capture hardware designs algorithmically and is compiled into a form used to configure the internal structure (function) of large custom ICs.

and everything else as TRUE. These instructions will only produce logical results, storing 0 for FALSE. Similarly, the continuous functions will interpret all arguments as continuous values, and produce continuous results.
The comparison instruction (>,<) are a hybrid in that these will treat the arguments as continuous numbers but produce logical results.
The conditional branch (IF) statement will interpret the condition as TRUE (take the branch) if the condition evaluates as non-zero. Upon taking the branch, execution will continue at the absolute line number supplied as an argument to this instruction.
By implementing both data types, the system can efficiently cover the input / output requirements of a typical embedded control application: The continuous data types would be used for reading numeric values from various sources i.e. to sample a continuous value such as temperature, position or time. The boolean type input / output can be used for logic level type sensing i.e. control switches, limit sensors etc.
In a typical embedded application, these two types would utilise different types of storage; boolean variables can be held in bit – wide memory (available with some microcontrollers) and continuous variables in byte, word or long-word sized storage. Unlike the simulation here, floating point storage (and mathematics) would usually only be used if strictly necessary - with continuous variables normally represented as integers, or bipolar integers (using two's complement representation). Storage size / representation minimisation in this way, is another example of how embedded applications are 'designed down to fit' [3.2] and this approach will optimise the design solution in terms of storage and speed: The use of bit and byte sized variable representation can allow highly – efficient one – to – one compilation of high level program statements down to machine – code primitives such as Branch On Bit Set.

4.5 Generating valid programs

During evolution, the crossover operator is forced to cut on word boundaries and the mutation operator is restricted to select from only permissible values based on the grammar and statement context. These restrictions ensure that no syntactically illegal programs can be generated,

4.6 Trial system implementation

In order to demonstrate the trial system, a LGP program evolution mechanism was required. There are many pre-written GP systems available for free download from the internet. Unfortunately, the provenance and quality of such free software is frequently unknown – and the consequent potential to waste effort great.

In order then, to pre – filter these free systems, only those available via the EvoNet web site[8] were considered. The assumption here was that these systems had been used by other researchers, and that these systems were, hopefully, 'tried and tested'.

The suitable choices were mainly written in Java for transportability, with alternatives written in SmallTalk and Ruby. The majority of systems were completed demonstration / experimentation environments.

At the time, this research may have taken the path of actually demonstrating the technique at the machine level with embedded hardware, and the operational speed of the system could have been an important issue in real-time control suitability. For this reason, systems not written directly in a language designed for low-level implementation on a ready platform, were ignored. Additionally, the computing resources were limited and this steered the choice away from any high level or interpreted languages whereby evolution time could be prohibitive.

This left the most suitable system to be GPkernel, a C++ class library of EC parts. However, two additional requirements for the evolution system included the ability to alter the system chosen significantly (in order to investigate language and data types specific to this research) and the facility to run concurrent tasks. The former reason made the use of a pre –written demonstration system unattractive because new, purpose – built software is frequently more reliable than massively – altered code.

Microsoft Visual C++ version 6, was chosen for implementing both the evolutionary system and running the completed control system later (in simulation) because of the multi – threaded application development path available with this tool. The production of a fast evolutionary system via the optimizing compiler, and low – level program development option afforded by

---

[8] http://evonet.dcs.napier.ac.uk/

using 'C', coupled with the good support available, cemented the choice upon this tool, along with the decision to develop a system from scratch.

4.7 Evolutionary process and parameters

This research is not intended to be a study into EC operation and the effect of evolutionary parameters on particular problems: Such work comparing crossover types, for instance, has been covered by others [Keijzer et al 2001][Menon 2002]. Thus, beyond the string length and population size variations made during experimentation, these parameter settings were not investigated but instead left at a useful working value found by trial and error. In fact, the system did not exhibit much sensitivity to changes made in these values - within the ranges studied.
Similarly, the mechanisms of EC used were basic and not researched: Evolutionary selection here creates a generational population of constant size. All candidates for the next generation are selected from the current generation by use of a tournament of size two.
The developed evolutionary system does, however, allow for the use of different population sizes, and mutation and crossover rates, for each task / output in case the ability to adjust the evolution process for each output / function became necessary. Individual parameter values per output would also facilitate forced solution diversity [2.7.3.11] if necessary later.

4.8 Methodology

Ideally the potential of this research would be demonstrated by capturing and evolving the solution for a real – world problem, and executing this on the intended hardware host under a multi – tasking operating system (OS). For the purposes of comparison, this solution could be contrasted to a solution created by programmers for the same problem. Suitable industrial partners were sought in an attempt to implement such a comparison - but the embedded applications of local companies (with links to the university) were safety – critical (e.g. traffic light control) making these wholly unsuitable targets from the outset.
Alternatively, the solution and implementation of a genuine, real – world control system could have been achieved by writing the proposed OS and

embedding this, along with the evolved code, into dedicated hardware, interfaced to the system being controlled.
However, this amount of work exceeded the time available for this PhD, and is arguably an inefficient use of time: The complete demonstration of the proposed system for one type of embedded control problem, would not prove its suitability to others – even within the target problem class of non – safety critical, concurrently - decomposable problems, that have relaxed requirements for output accuracy and response. Instead then, a more pragmatic and realistic approach was adopted here to demonstrate the potential of the proposed technique by:

(1) Solving a simplistic though representative, multiple – input, multiple – output control pilot problem (the Fridge controller) by separating the entire problem into a two input, three output problem that uses two data types (Boolean and bipolar continuous integer).

(2) Using the knowledge gained in the solution of the above pilot to solve a different representative and more complex problem involving 10 inputs, six outputs and two data types (the Washing Machine controller).

(3) Providing a simple, visual demonstration of the operation of the evolved Washing Machine controller in operation.

4.9 Example problems

The example problems chosen to demonstrate this technique were a simple fridge controller and later a washing machine controller: Both are familiar examples of everyday control problems that do not require any prior specialist domain knowledge: Both fit the criteria above [3.7.2] as suitable targets; i.e. non safety – critical, do not require great accuracy or have critical response timings; and the software would not generally be regarded as 'challenging' or 'stimulating' for a human programmer. Finally, the two problems indicate how the technique is expandable i.e. the fridge problem has three outputs / tasks and two inputs, whilst the washing machine controller has six outputs and ten inputs.

4.10 The Fridge controller

The Fridge controller problem [Hart & Shepperd 2002] is a relatively trivial example of how a control system can be decomposed into a number of parallel tasks - each with a separate output - and how the code producing those outputs may be created. The problem requires control code that will simultaneously monitor two inputs and produce three outputs of different types.

The Fridge problem is included here only to give a simple introduction into how the decomposition translates into a real world application. The Fridge controller is also included to show how the technique can be scaled to cope with more complex problems i.e. the Washing machine controller, to follow.

4.10.1 Fridge controller inputs

4.10.1.1 Temperature.

The temperature inside the fridge is assumed to be sampled and resolved to an eight bit data value. This is a continuous-type input and, for a real implementation, this analogue temperature value could be generated using a thermistor (temperature-sensitive resistor) and an Analogue to Digital Converter (ADC).

4.10.1.2 Door status.

This is a boolean logical - type input that, in a real implementation, could be read as one bit from an external input port. This input will be logic 0 if the door is open.

4.10.2 Fridge controller outputs

4.10.2.1 Alarm.

This logical output will be set if the Temperature inside the fridge exceeds the maximum allowed temperature. It is assumed that the imaginary fridge contents will provide sufficient hysteresis (due to thermal mass) to prevent sudden fluctuations in temperature.

4.10.2.2 Fridge light.

This logical output will drive the internal fridge light on when the door is open.

4.10.2.3 Pump drive.

This is the continuous refridgeration pump drive output. For the purposes of this example, the fridge will be driven cooler when the refridgeration pump demand is increased; therefore the pump drive is directly proportional to the internal temperature. In a real fridge, this would usually be under closed – loop control i.e. the drive would respond to the temperature error (between the desired and actual internal temperature), Here the control is open – loop i.e. there is no feedback to produce an error term.
The fridge temperature achieved would be related to the drive by a number of factors including the electrical, hydraulic and mechanical characteristics of the pump and interface; the efficiency of the fridge, the volume of the fridge, the ambient temperature, disturbances to the system (door opened) etc: However, the fridge problem was primarily intended as a vehicle with which to introduce this research, and so a simple function was arbitrarily chosen to represent the pump transfer function:

Drive = (Temperature + 10) * 2

4.10.3 Fridge control functions to evolve

For this example, we shall assume normalized and scaled inputs i.e. one temperature input unit equals one degree Celsius : Thus one possible solution for the fridge problem, and hence the software to control the fridge, could be:

```
Out1 = NOT In1
Out2 = In0 > 5
Mem0 = In0 +10
Out0 = Mem0 * 2
```

or some equivalent that would express that the light is on when the door is open (Out1); that the alarm output should be true when the temperature exceeds 5 degrees (Out2) and that the fridge pump demand (Out0) needs to be driven proportionally in relation to the temperature.

4.10.4 Solving the fridge problem

To evolve the Fridge controller, the problems encountered, and changes to the technique adopted in response, are published in detail as [Hart & Shepperd 2002]. All of these points are covered in the following chapter relating to the Washing Machine controller. For quick reference, the parameters used to evolve the Fridge controller are outlined in Table 4.1 below:

Parameter	Value
Population size (fixed)	200
No. of demes	3
Representation	Variable length LGP String
Selection mechanism	Tournament
Crossover rate (1-point crossover)	0.2
Mutation rate	0.1
No. of generations	30000 (max)

Table 4.1: Fridge controller evolution parameters

<!!In summary, tackling the Fridge problem led in two main conclusions:

- That a multiple – input, multiple – output control problem, using two data types, can be successfully solved using EC and appropriate problem decomposition.

- That the system was prone to premature convergence after only one output was solved – effectively removing population diversity that could be necessary to solve the remaining outputs [5.2].

The encouraging outcome of the Fridge controller pilot led to the larger, more complex. multiple – population Washing Machine controller problem. The Washing Machine controller, detailed below, supersedes all aspects of the Fridge problem and its solution, consequently the remainder of this document will focus on that problem alone.

4.11 The Washing Machine controller

This second study is an extension to the previous pilot - the Fridge controller - in that it targets a more complex problem with seven threads and introduces temporal sequencing rather than continuous operation. Again, the subject falls into the target class of problems identified in [3.7.2] i.e. that the application is non safety – critical and does not demand high accuracy in terms operational precision or response timing.
The basic washing machine control program here will perform the following operations:

Regulate the water temperature

Control the cold and hot water fill and drain operations

Control the drum rotation speed

Sequence the wash, rinse, drain and spin operations

4.11.1 Inputs and outputs

In the real Washing Machine controller, these would form the electronic interface between the microprocessor components and the rest of the washing machine: These signals would typically require appropriate level or type conversion i.e. from voltage to current, low voltage to high voltage etc:

4.11.1.1 System inputs

Timer inputs TS0 .. TS6

The temporal sequencing of the entire wash operation is driven by the seven binary timer inputs TS0 to TS6. These inputs are mutually exclusive in action and in a real application using the general purpose hardware block described in [3.10] would be sourced by the embedded OS. Each input will be set active for a time period suggested by the designer via the front end tool, and then relinquished before the next timer input in sequence is set active. The duration of each operational timeslot will then reflect the work to be done i.e. in our example here, the Wash period would be longer than the Drain period.

Target temperature.

This continuous input provides the control system with the required washing temperature for the water in the drum. This will be in the form of an integer from 0 to 90 and is derived via the interface electronics from a dial on the front of the washing machine.

Water temperature

This is a continuous reading of the drum water temperature and here the value supplied will be an integer in the range of 0 to 100.

Drum Full

The remaining input here is the Drum Full signal which will become active when the washing drum is sufficiently full of water.

4.11.1.2 System outputs

There are a total of six outputs driven by the software being cycled in seven tasks by the OS:

Heat.

This is the drive for the drum water heater and the objective here (as indicated by the product designer) is to regulate the water temperature to that required for the Wash cycle. The target washing temperature is

set by the Target temperature control input above. The aim is to evolve code that will take the water temperature error (Target temperature minus actual Water temperature) and multiply this by a constant to match the full span of an eight bit number for byte – wide interface with the heater control. In this way, the resolution is optimised for the hardware employed so that the maximum temperature error expected will result in the maximum output drive.
In addition, a secondary requirement is that if the Water temperature exceeds the Target temperature then the output is forced to zero.
The overall Heat output is actually formed by combining the output of two tasks – one producing a logical output and the other a continuous output - to ease evolution [4.1].

Hot, Cold

These two boolean outputs control the hot and cold water feeds into the washing machine drum.

Slow, Fast

The drum rotation motor is controlled by these two boolean outputs so that if Slow is active the drum will rotate slowly for washing, and Fast will command a fast rotation for spin – drying. The drum motor will not turn in the absence of both signals, and spin fast if both are set to '1'.

Drain

Water will be pumped from the drum when this boolean signal is active.

4.11.2 Complete operational cycle

In figure 4.1 below, the desired interrelationships between the various inputs and outputs are shown:

Figure 4.1: Relationships between inputs and outputs for the washing machine.

	TS0	TS1	TS2	TS3	TS4	TS5	TS6	DrumFull
Heat (cont.)		f(Te)						
Heat (logic)		x						
Slow		x	x		x	x		
Fast				x			x	
Cold	x				x			x
Hot	x							x
Drain			x			x	x	
	(Fill	Wash	Drain	Spin	Rinse	Drain	Spin)	

where f(Te) indicates a function of water temperature error.

4.11.3 Functions to be evolved per output

The following demonstrate the functional relationships that could be evolved to satisfy the training requirements and hence the overall washing machine problem. Functions actually evolved may, perform the same function but by different though equivalent programs. Alternatively, the system may evolve solutions that are not correct - though satisfy the training and test sets - due to ambiguities there. This would be due to an incorrect specification of the problem via the training and test sets generated here by the front end tool.

Task 0: Heat (continuous)

Heat = (Target temp - Water temp) * Span constant

Task 1: Heat (logic)

Heat Gate ON = (Tt > Tw) & TS0

Task 2: Motor Slow

Slow = TS0 | TS1 | TS3 | TS4

Task 3: Motor Fast

Fast = TS2 | TS5

Task 4: Hot In

Hot = TS0 & ! Drum Full

Task 5: Cold In

Cold = (TS0 | TS3) & ! Drum Full

Task 6: Drain pump

Drain = TS1 | TS4 |TS5

4.12 Chapter summary

This chapter outlined the demonstration system, and the EC techniques and tools used. Two sample problems are detailed, and the larger problem – the Washing Machine controller – will form the main focus of the next chapter: This will include details of the problems encountered in realising the demonstration, and the solutions tried and adopted. Lastly, the results of some 900 experimental runs will be presented; followed by an examination of these results, and how these may point to an alternative evolution strategy.

5 Experimentation

5.1 Introduction

The previous chapter detailed the method used in this work to demonstrate how the software for certain control systems may be created automatically. This chapter examines the problems encountered, and the changes made, to realise this goal: These problems centre upon improving the search results by attempting to avoid the premature convergence of the population.
Later, the results of 900 evolutionary runs involving the most complex task are supplied; and an examination into the effect of population size and Linear Genetic Programming (LGP) string length on search performance, are made: Following on from this, the option of restarting from generation zero against continuing with the current run - in the absence of search progress - is considered.
Lastly, an operational simulation of the entire evolved, washing machine is described, along with the LabVIEW graphical application used to visualise the wash operation.

5.2 Technique evolution: One population, multiple outputs

The early attempts at evolving control programs were made using a single population with the requirement that the solution would generate all system outputs - this is in contrast to using one single, dedicated population for each task / output posited earlier. The single population / multiple outputs method was used when tackling the fridge problem [4.10]: Here, the LGP system would typically manage to evolve solutions satisfying the requirements of one or more outputs but generally failed to find solutions for the remaining output(s). This appeared to be a result of premature convergence, and as a consequence, the population diversity was lost. This loss of diversity resulted in a loss of problem – solving potential as the other solutions started to emerge. Diversity could then only be reintroduced by means of the mutation operator, but the effect of this was restricted due to the similarity of the population due, again, to the convergence.
Solving for several outputs within a single population in this manner is essentially a problem of Pareto optimisation. Methods for producing such multi – objective solutions have been previously researched, leading to the

adoption of multi- population techniques such as Clients and Civil servants [Ryan 1999]. Here, N populations are targeted at N specific problems and a composite solution is found by combining useful elements from the solutions provided by each population.
One of the tenets of this work was to assume computing resources should not be a limitation – within reason. This line of thought makes the adoption of a one output per population / task approach acceptable: This method will result in a reduction in the evolutionary search burden (per population) whilst making the technique scalable to more complex problems i.e. problems with more output channels in this instance.

5.3 Cumulative fitness assessment

The fitness assessment method originally employed to score each member of the population was simply to calculate the absolute error between the idealised training set output vector value, and that provided by the candidate solution, for the same input training vector value. The reciprocal of this error term was then used as the fitness value for that training vector case (such that the fitness is directly proportional to the accuracy of the candidate solution for that training case). The overall string fitness value scored was then simply the accumulation of all such fitness values for all the training vector cases.
This cumulative fitness assessment method proved to be unsatisfactory with a tendency toward premature convergence: Frequently the system would simply find a solution that would force a constant into the output register that matched at least one output training vector value. Similar solutions with higher fitness were sometimes delivered when conditional jumps were evolved to examine the input vector value and so deliver two different constants to the output register.
Once such an operation appeared in the candidate solution, the population would start to converge and the situation was unrecoverable – even if the system was left to run for one million generations.

5.4 Preserving diversity: Qualified cumulative fitness

Generally EC techniques are used to tackle problems where the 'correct' or 'best' answer isn't known: In this situation, the population needs to converge as an indicator of success; however, this convergence inevitably leads to a

loss of diversity in the population. The initial random population, created for generation zero, should offer the greatest variety of programs that may form the root of a solution.
To prevent the system from placing constants in the output register (and so start to converge) a new fitness assessment method was tried: The evolved program string would be scored as having zero fitness, unless it exhibited a threshold amount of fitness for each training vector case. Provided this criterion was met, the fitness score for the string would again be the summation of the per case fitness as before.
This method was adopted because, in the case of functional regression of continuous functions, the solution should grow from a useful 'root' version of that function; such a 'root' solution is more likely to exhibit some fitness in all training cases. Delaying any form of convergence until such a useful root function is discovered, preserves the initial start-up diversity (of the initial population) for longer.
One alternative strategy for allowing the necessary convergence to occur, whilst potentially maintaining some of the initial overall diversity, has been achieved by the use of island - model parallel Genetic Programming (GP) [Fernandez et al 2000]. Here, sub – populations converge - but this convergence maybe upon different (potentially sub - optimal) solutions. The exchange of material between these sub – populations can thus preserve overall diversity longer, allowing an optimum solution to emerge.

5.5 Fitness clamp

The use of a fitness assignment method that demanded a threshold fitness in all training vector cases, proved to be a more successful approach than the simple cumulative fitness assignment, in that the ideal solution was found more frequently. However, the calculation of fitness as the simple reciprocal of the error led to the possibility that as any evolved solutions approached the ideal, the error would fall to near zero, and the fitness would consequently tend toward infinity. By this process, a function that was very fit in just one training vector case (whilst being only slightly fit in all other cases) could swamp the population. This could mean that a better, 'all round' solution that was evolving would be passed over.
To avoid such a possibly unhelpful saturation of the population by a single candidate, a maximum fitness 'ceiling' value was employed. This clamp value

was used as the candidate's fitness score when the error for the training vector case fell below 0.002. This value was found to be appropriate for the problem at hand by experiment. The tolerance of this value did not seem to be too critical, but the choice for such a value must depend upon both the type of function sought i.e. its complexity, and the nature of the fitness landscape surrounding this solution. In a future development of this system, this value may well be tuned dynamically according to search progress [6.6.1].
One immediate side effect of clamping the maximum fitness score in this way, is to create a 'dead zone'. Within this region, solutions could not be improved upon because the feedback path that leads solutions to improvement through evolution is broken. This potential limit on solution accuracy can be seen as acceptable because of the targeted problem class of non safety – critical applications that do not demand high accuracy [3.7]. An example of an acceptable application with an accuracy tolerance would be a system to tune a radio or guide a TV satellite dish; in the radio case, the broadcasts are made centred upon the advertised station frequency and, as a consequence, tuning close to that frequency will still yield a good signal.

5.6 Fitness gain and root solution capture range

Demanding that any solution to be classed as a 'root' solution exhibits threshold fitness in all training vector cases applied, will create a solution 'capture range': Due to the reciprocal nature of the fitness assessment function, the errors seen have to be relatively small to in order to generate a significant fitness value. As this value must be sufficient in all training vector cases, then an acceptable candidate solution needs to be quite close to the ideal function before it is will be scored. However, the chances of creating such a close match to the desired function randomly are small, and, as a consequence, some tolerance on the size of this capture zone needed to be incorporated. This is provisionally achieved here by simply multiplying the fitness score by a constant.

5.7 Dynamic fitness gain

As an experiment to investigate the possible improvement in results that may be obtained, the fitness awarded to a given individual was allowed to vary according to search progress. The idea here was to increase the fitness by multiplying the award by a constant that was increased incrementally over evolutionary time if the search was failing.

It was hoped that the effect of gradually increasing the fitness award would eventually lead to individuals with poor fitness, but some potential, increasing in frequency in the population. With this increased presence, the chances of adapting a root solution constructively would increase leading hopefully to the optimum.

Unfortunately, if the initial population had no useful root solutions, and adaptation failed to create any, then the effect of this dynamic fitness allocation was detrimental: The effect is to eventually make candidates that placed one or more constants in the output register, appear fit enough (exceed the fitness threshold described in [5.4] above) in all training cases to start a fruitless convergence: The use of a dynamic fitness gain was abandoned.

5.8 Parsimony

The initial version employed a second level of fitness scoring whereby an additional, though minor, award was made according to the length of a candidate.

This simply involved increasing the fitness by one for each statement that the candidate contained that was less than the maximum number of statements permissible per string.

Applying parsimony to GP populations has been employed in an attempt to reduce program bloat [2.7.3.2] however this is less of an issue with the LGP form employed here, mainly because arbitrarily large subtrees cannot be grafted onto existing large trees.

Parsimony intends to reduce solution length but this will frequently reduce the population diversity too, hampering the search. For this reason, the true relevance of parsimony to LGP - and remembering that resources are not an issue in this work - parsimony was abandoned.

5.9 Fitness function used

In the above sections [5.3 to 5.8] the development of the fitness assessment method that proved to be the most successful is explained. In summary then, the fitness assessment algorithm applied to each population member during the experimentation was as follows:

(1) Load the input registers with the corresponding values from the input training vector.

(2) Clear all output register for the output associated with this population and execute the candidate string.

(3) Calculate the error in the output by subtracting the ideal (output training vector) value from the value in the relevant output register.

(4) Make this error value positive if negative.

(5) If the error value is less than the clamp threshold value (0.002) [5.5] then calculate the reciprocal of this error and multiply this value by 10 [5.6]; otherwise, set the fitness value to the maximum 'clamp' value (5000).

(6) Repeat steps (1) to (5) for each training vector.

(7) If the fitness for any training vector applied is less than the threshold (0.2) then set the fitness value for the whole string to zero [5.4]; otherwise, sum the fitness score for each training vector and assign this to the string.

5.10 Crossover

Crossover action on LGP systems can be viewed as simply a form of strong mutation: With register-based linear GP, it will make little difference (in many cases) in which order unrelated program statements are executed: Thus the action of exchanging the top and bottom halves of two strings, in a converged population can add little. However, this ordering is certainly more critical in the case of the Output statement: In this case, the sequential execution of the program strings (unless branching occurs) creates a situation whereby only the final Output statement executed (and the code that built the data for this) is important. Perhaps not unexpectedly then, the level of crossover used has little impact on the level of success attained.
The crossover rate used was held at 0.25 i.e. on a pass through the population, the probability of a particular string being selected for crossover

was 0.25. When two strings have been selected in this manner, a random cut point is chosen separately for each string. If the crossover action would produce a string whose length exceeds the preset maximum, then the operation is abandoned; otherwise the two string sections above the cut point are exchanged, and the adapted strings returned to the population.

5.11 Mutation action

Subsequent to the initial population, the only way of introducing diversity into the population is through the action of mutation. Mutation becomes especially important then as the population starts to converge – especially if that convergence is upon a sub-optimal solution.
Various methods of mutation were tried, along with the replacement of the least fit individuals in the population with an equal number of randomly created strings.
Though many methods of mutation were tried, the final method used in the experiments formed a composite of these: The maximum evolutionary time allowed is 20000 generations[9] and this time is divided into four epochs, with the level or type of mutation applied changed according to the current epoch:

> During epoch 0 which lasts for the first 40% of evolutionary time, one single statement is chosen randomly from the string selected for mutation. This entire statement is replaced.
>
> During epoch 1 which lasts from 40% to 70% of evolutionary time, the mutation action will again be to randomly select a single statement from the selected string but now only one part of that statement will be altered:
> this will be either the operation type, the output channel type or one of the two input channels [see appendix A: Evolutionary language grammar].
>
> During the third section of evolutionary time (70 to 90%) one section of the chosen string will be selected and, if that section contains a constant value, then this will be altered. In this instance, only constants

[9] N.B. Evolution can be terminated early if the mean population fitness exceeded a pre-set value i.e. when the population has converged upon a high fitness solution.

in the input channels are modified – if used. Unlike the mutation of constants above, the current value is retained and updated by the addition of a bipolar gaussian random number. In this epoch, this number is randomly picked from a gaussian normal distribution with mean 0 and standard deviation 2 [see appendix B; Gaussian random number generation].

In the final epoch, which can run from 90% until the end of evolutionary time, an identical process to that above is applied - except that the random value added to the existing constant is now picked from a distribution with a standard deviation of 1 – effecting a smaller change on the current value (probabilistically) than in the previous epoch.

The intention of performing the mutation in this way is to focus down upon the candidate solutions prevailing in the population at that time. The intention is that as evolution progresses, those candidate solutions will become closer and closer to the ideal solution and so require less dramatic changes. For the final two epochs, any input constants are the only items that are available for change, and the size of that change to their current value will reduce.
The focussing down of the search area over time is expected to produce a similar effect to that of Simulated Annealing [2.4] but this method also dates back to the earliest days of evolutionary programming and Fogel's work with Finite State Machines [Fogel et al 1965].

5.12 Population size

Initially a population size of 200 program strings appeared to be a good compromise between computing time and efficacy during development. Though there has been published work that attempts to relate population size, representation and convergence [Albuquerque, 2000]. Albuquerque suggests that there exists an optimum population size for a given representation; too small and there will be no convergence; too large and computational effort will be wasted – perhaps pointing to a multiple population approach as the ideal? Unfortunately Albuquerque's formula for optimum population size is particular to the Generalised Simulated Annealing algorithm used, and the representation adopted – and not in a form generally applicable to other EC contexts.

5.13 Training set size

At just sixteen cases per output, the size of the training set used in the experiments is unusually small for EC: However, because the fitness function adopted here requires a threshold fitness score in all cases, then the candidate function promoted is most likely to be a useful root function which can map all training cases to some extent: By this process then, a relatively small training set was capable of unambiguously describing the target function (or its near equivalent) in the example tried.
In the absence of the front-end tool discussed in [3.9] the training vectors were created manually: Sixteen cases per output proved to be sufficient (during development trials) to unambiguously describe the required functions to be evolved: The size of training vector required for this system will have to balance the following requirements:

- The evolution time is directly proportional to the size of the training set because fitness assessment of the candidates will usually involve testing that candidate against each training case. The time taken to evolve a solution can be an important consideration where computing resources are restricted, or where evolution time must be restricted: This leads to pressure to minimise the training set size.

- The training vector size has to be large enough to explicitly relate the correct inputs to the target output and describe the functional relationship necessary. This is vital because in the proposed system, all tasks / populations are exposed to all system inputs during output evolution, and so it is possible for an incorrect relationship to be discovered between the wrong inputs and an output: This can be countered by ensuring that the training vector size is large enough to include cases whereby a coincidental relationship between inputs and an output isn't inferred.

- The fitness assessment method used [5.4] requires that candidates exhibit threshold fitness in all training cases before being awarded any fitness score: This is intended to promote candidates offering a functional relationship between the inputs and outputs rather than outputting constant values. Search success will involve the capture and improvement of useful root solutions that fit this criteria initially, however, as the training set size increases, discovered functions will need to be nearer to the ideal to satisfy this criteria, thus the capture range will reduce, and the search will need to have a finer focus – in turn reducing the probability of success.

- More complex functions will need larger training sets for unambiguous description.

- It is possible to exactly define all the required functions with a minimal training set – if the required functions are simple.

Training vector size and generation is further discussed in [6.3].

5.14 Composite outputs

The requirements of the heater demand output [4.11.1.2] combine tracking the error in drum water temperature, cancelling this demand if this error is negative (i.e. the drum water exceeds the selected wash temperature) and ensuring that any heating only occurs in the correct timeslot.
This combination of one continuous function and two logic functions together proved to be extremely difficult to solve for an LGP type system. Typically, the logical parts of the problem were solved but the incorporation of these with the required continuous function remained hard.
The explanation for this is probably that the Stop / Go texture imposed on the fitness landscape by the logic functions effectively obscures any tractable gradient necessary to find the outstanding continuous portion.
The solution employed is to separate the logical and continuous sections of the problem and combine the results afterwards by a 'gating' operation. The gate control would be fed by the logical contribution and the effect would be to either pass the continuous contribution untouched or clamp it to zero.
Arguably, this could be seen as a solution to the particular problem at hand, whereas this research suggests a general - purpose program creation technique that may be applied to any problem in the identified class. To make this solution universal then, all continuous outputs would have an associated logical gating function: The gating itself would occur in the embedded Operating System (OS) controlling and cycling the evolved, tasks. If there were

no logical / continuous mix needed, then the associated logical gating control task would be undefined, and the gate itself would default to 'open'.

5.15 Results

The useful evolution of the majority of the tasks proved to be quite trivial with maximally – fit solutions found in less than 20000 generations (about half an hour of computational time on a 1 GHz PC).
Evolving task code that produces continuous rather than boolean output, proved to be more problematic with good solutions only being delivered around one in every sixteen runs. In order to examine system performance then, the results gathered over 900 runs record only the evolution of the continuous Heat function.
The experimental results are summarised below: The maximum fitness score attainable is 80000 and a score in excess of 40000 would typically indicate that the correct function had been induced - but that a further refinement in constant values could be made: For example, if the function sought was

$$O = (In.1 - In.2) * 2.71$$

And the best candidate at a given time was

$$O = (In.1 - In.2) * 2.6$$

Then this function would score above 40000 here: Leaving the EC system running longer would eventually result in a refinement of the constant, up to the required 2.71 value, and a fitness score near 80000.

The first two tables below [5.1 and 5.2] show search successes as the population size is varied. In the first table [5.1] 'success' is awarded for fitness in excess of 40000, and fitness in excess of 75000 in the second table [5.2]. Similarly, the third and fourth tables below [5.3 and 5.4] record success at 40000+ and 75000+ levels against varied initial (randomly generated for generation zero) and maximum permitted string lengths:

Pop:	200	400	600	800	1000	Totals
No:	170	170	169	167	168	844
Yes:	10	10	11	13	12	56
Totals	180	180	180	180	180	900

Table [5.1]: Success (40000+) v. population size.

Pop:	200	400	600	800	1000	Total
No:	172	173	171	167	168	851
Yes:	8	7	9	13	12	49
Total	180	180	180	180	180	900

Table [5.2]: Success (75000+) v. population size.

Length:	5	10	15	20	25	30	Total
No:	137	138	144	142	142	141	844
Yes:	13	12	6	8	8	9	56
Total:	150	150	150	150	150	150	900

Table [5.3]: Success (40000+) v. string length.

Length:	5	10	15	20	25	30	Total
No:	138	138	144	146	143	142	851
Yes:	12	12	6	4	7	8	49
Total:	150	150	150	150	150	150	900

Table [5.4]: Success (75000+) v. string length.

5.15.1 Evolvability and string length

Using longer strings to construct an initial population would generally result in a more diverse population and, intuitively, this should increase the chances of finding a good root solution. However, the results [tables 5.3 and 5.4] indicate the opposite result – that greater success results from shorter strings. There would seem to be two possible explanations for this unexpected result: Firstly, the evolved program strings are executed sequentially forwards (with the exception that the Branch instruction may cause some instructions to be skipped over) and so the contents of the output register will be overwritten by successive output operations. As a consequence, only the code and data directly contributing to the final output operation is relevant; thereby the overall string length cannot be simply viewed as an indicator of potential.

Secondly, the effect of mutation (and effective search) are diluted in direct proportion to the string length. This is because the probability of selecting the critical instruction for a constructive mutation must diminish as the string length increases.

These two factors can explain why the counter – intuitive observation that string length is negatively related to success.

5.15.2 Evolvability and population size

Similarly and perhaps more surprisingly, the experiments also failed to demonstrate any useful advantage in operating with larger population sizes. Again, the greater diversity afforded by a larger initial population should increase the probability that a useable root solution will appear in the initial random population. However, it has been shown that, beyond a given threshold, the functional diversity of a random population will tend to a limit irrespective of size [Langdon 2003], but perhaps the dominant factor here is the subsequent loss of diversity:

Due to the fitness assessment measure used [5.4] the vast majority - or possibly all - of the candidate solutions in the initial population will be scored at zero fitness. As a result, approximately half of the diversity can be lost on selection, and this process will continue with the population starting to converge upon any solution with some fitness.

Unfortunately, premature convergence on any sub – optimal solution in this manner, will effectively block the development of any potentially ideal solution (uncovered by mutation or crossover) unless the fitness of this root solution is sufficiently high.

This unexpected indifference to population size (and sometimes other genetic parameters) in some situations, was also noted and investigated by Fuchs [Fuchs 1999] in a study where GP and Hill Climbing (HC) were compared. Fuchs' conclusion was that this indifference is more an artefact of the particular problem rather than the technique (i.e. another example of the No Free Lunch theorem [Wolpert & Macready 1995]). Fuchs suggests that in such a situation, the fitness landscape might be flat and (initial) solutions could only be found by 'accident': In such a situation, a GP search with a population size of one, could be seen as a basic HC search (where the HC can restart at a random point if no progress is made): The conclusion then, is that GP population size can be irrelevant in some situations.

5.15.3 The rate of random generation of root solutions

In an attempt to determine whether or not the action of premature convergence will block the development of good root solutions - irrespective of (reasonable) population sizes - the rate of generation of such root solutions was examined: The results are lists of the fittest, unadapted individuals that were generated randomly as members of a population of variously – sized strings. These lists were created by modifying the standard evolution software used previously, so that rather than adapt and evolve a random start – up population over 20000 generations, a fresh random population of 1000 programs were created 20000 times. Each of these populations were examined and the fitness of any fit strings added to the list, along with the population number (i.e. 1 to 20000). Due to the amount of data, the content of these lists is supplied as a histogram and included in Appendix C.

The lists demonstrate that the frequency of occurrence of fit individuals is so low (at best 44 individuals of fitness 116.9) that counteracting any prior (sub – optimal) convergence would be unlikely, due to the relatively low initial fitness of these individuals.

Also, by examining these lists of fittest individuals produced during 20000000 attempts, where, in the best case, the highest fitness recorded is 116.9, it may also be concluded that the LGP system used is a very effective method - capable of producing solutions with a fitness in excess of 75000 on approximately one run in every eighteen [tables 5.2 and 5.4]. The results are slightly better still (56 / 900) for successes with a fitness above 40000 [tables 5.1 and 5.3].

5.15.4 Continued evolution versus restarting

Due to the inevitable convergence, and low initial fitness of solutions (as discussed in [5.15.3] directly above) continuing with a run that is not making useful progress may be a waste of resources: An alternative strategy may be to abandon the run after a time and simply restart from zero. To assess the merit of this approach, the data from the 900 experimental runs above was examined and the epoch in which the root solution first appeared (fitness left zero) was recorded:

Epoch	Generation		#Runs
0	0..8000		486
1	8001..14000		127
2	14000..18000		13
3	18001..20000		5
Zero fitness			269
		Total:	900

Table [5.5]: Epoch of discovery for fit individuals.

For the 49 out of 900 totally successful (i.e. fitness of 75000 and above) runs alone, the epoch of origin was as follows:

Epoch	Successes	Probability	
0	32	32/49	65.3%
1	15	15/49	30.6%
2	2	2/49	4.1%
3	0	-	-

Table [5.6]: Epoch of discovery for successes.

So the probability that one of the successful solution will emerge during the first epoch is 65.3% and this is only the first 40% of the total allocated computational effort allocated for that run.
Alternatively, the probability of discovering and building a solution to success during the remaining epochs is:

$$(49 - 32) / (900 - 486) = 4.1\%$$

And the probability of finding a successful solution during the first epoch is:

$$32 \ / \ 486 \ = 6.6\%$$

Thus, with this data, there appears to be a clear advantage in restarting after the first 40% of a run - if the fitness has not left zero by this time.

This higher rate of discovery (of initial root solutions) during the first epoch is almost certainly due to the more radical mutation strategy used in this epoch [5.11].

5.15.5 Testing

Some initial simulation work was performed on the evolved washing machine code by synchronising and then cycling the evolved threads in parallel to simulate the action of the embedded OS: In this simulation, sensor data was read in from an input file (replacing the data from the system hardware) and this was run through the code threads to produce the output data, which was collected synchronously and written to an output file (instead of writing to the physical hardware). This collection of input data samples and corresponding output data was then used as input for a LabVIEW[10] application written to graphically display the simulated washing machine state.

The LabVIEW application converts the input, and corresponding output file data from numeric data and displays it as actions in the washing operation; advancing to the next data batch / washing state as the file data is stepped – through. The result is a display that could show, for example, spin speed and water temperature, at a glance.

However, this graphical simulation is intended only to give a manual demonstration of the operation of the whole system and there remains great scope for further work [6.6.3] utilising automated test vector generation. The effect of intensive automated testing could be to raise confidence in the technique and so increase the range of target applications.

Ultimately, graphical simulation of the evolved system could be used for prototyping and developing the design prior to implementation, QA, sales demonstrations etc.

---

[10] LabVIEW:
LabVIEW is a 'graphical' programming language used primarily for fast application development in the electronic test and automation industries. Graphical icons (depicting a function) replace program statements and data flow is created by connecting these icons with virtual 'wires' - which appear as lines. Control flow and sequencing is determined by data flow and by grouping icons in operation boxes for iteration operations or for the purposes of modularization. Various forms of numeric data can be easily displayed with the use of various library icons such as graphs, meters or fill bars. Similarly, data can be input using virtual 'knobs' or via disk files or networks – again using library icons. For more information on National Instruments' LabVIEW: www.ni.com.

5.15.5.1 LabVIEW simulation screenshots

The following figures [5.1, 5.2, 5.3 and 5.4] are screenshots taken from the LabVIEW simulation of the entire washing machine controller operation.

In these figures, the upper block of boxes show the sensor input fed to the controller from a data file: The Vectors input shows the total number of system input and output sample vectors to display. The Time box indicates the current operational timeslot; the Water temperature and Target temperature boxes indicate the actual and required drum water temperatures, and the Water level box corresponds to the Drum Full input.

The lower five boxes display the collected system outputs taken from the evolved code threads when fed with the sensor input data: The Drum speed indicates the washing drum rotation speed, the Heat demand shows the composite logical and continuous Heat demand values; the Drain pump box indicates whether or not the drum water drain pump is operating, and the Water boxes show the status of the hot and cold water feed valves.

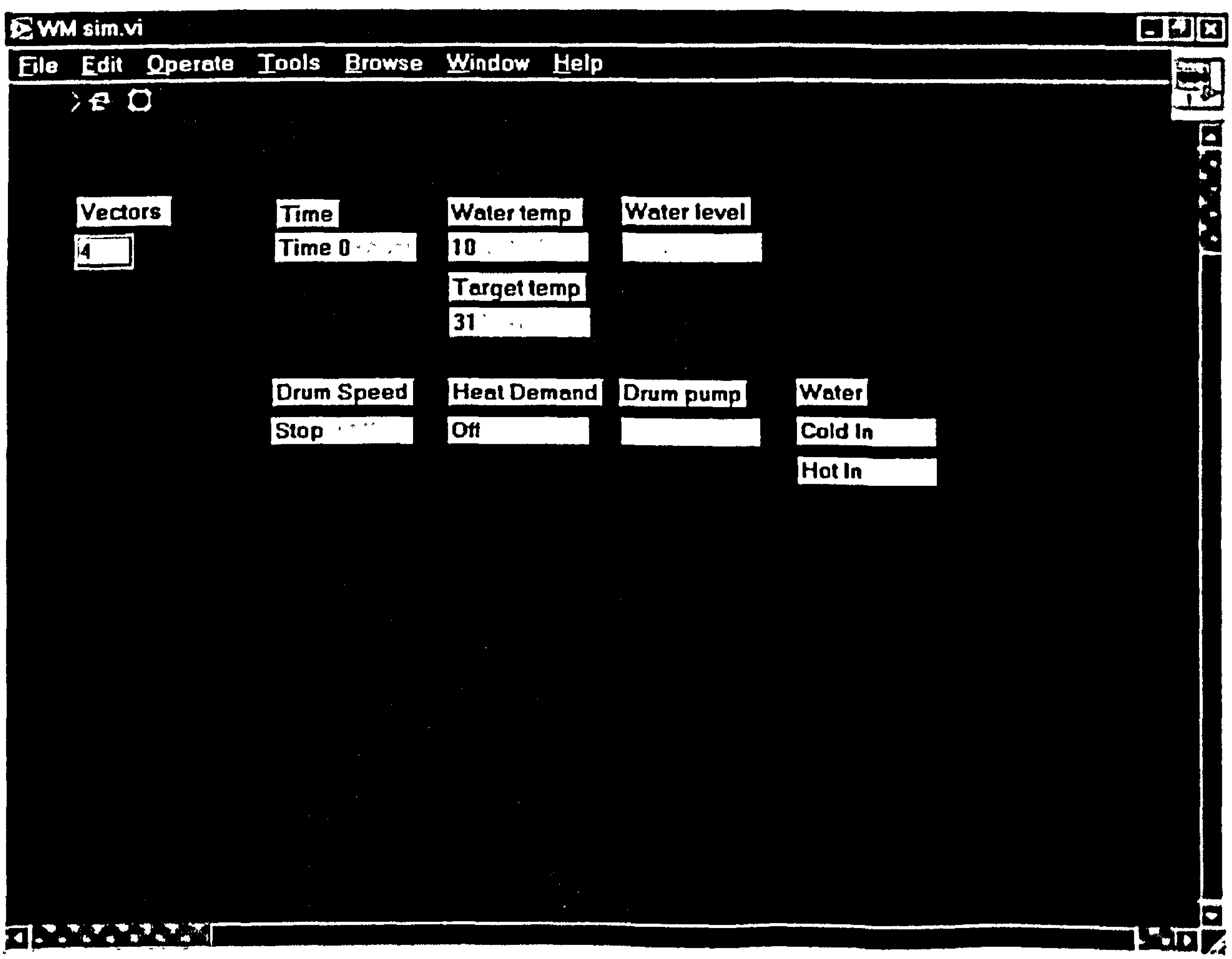

Figure 5.1: Timeslot 0 (Fill). Here the drum is filled with hot and cold water, tho drum is not rotating and there is no water heating.

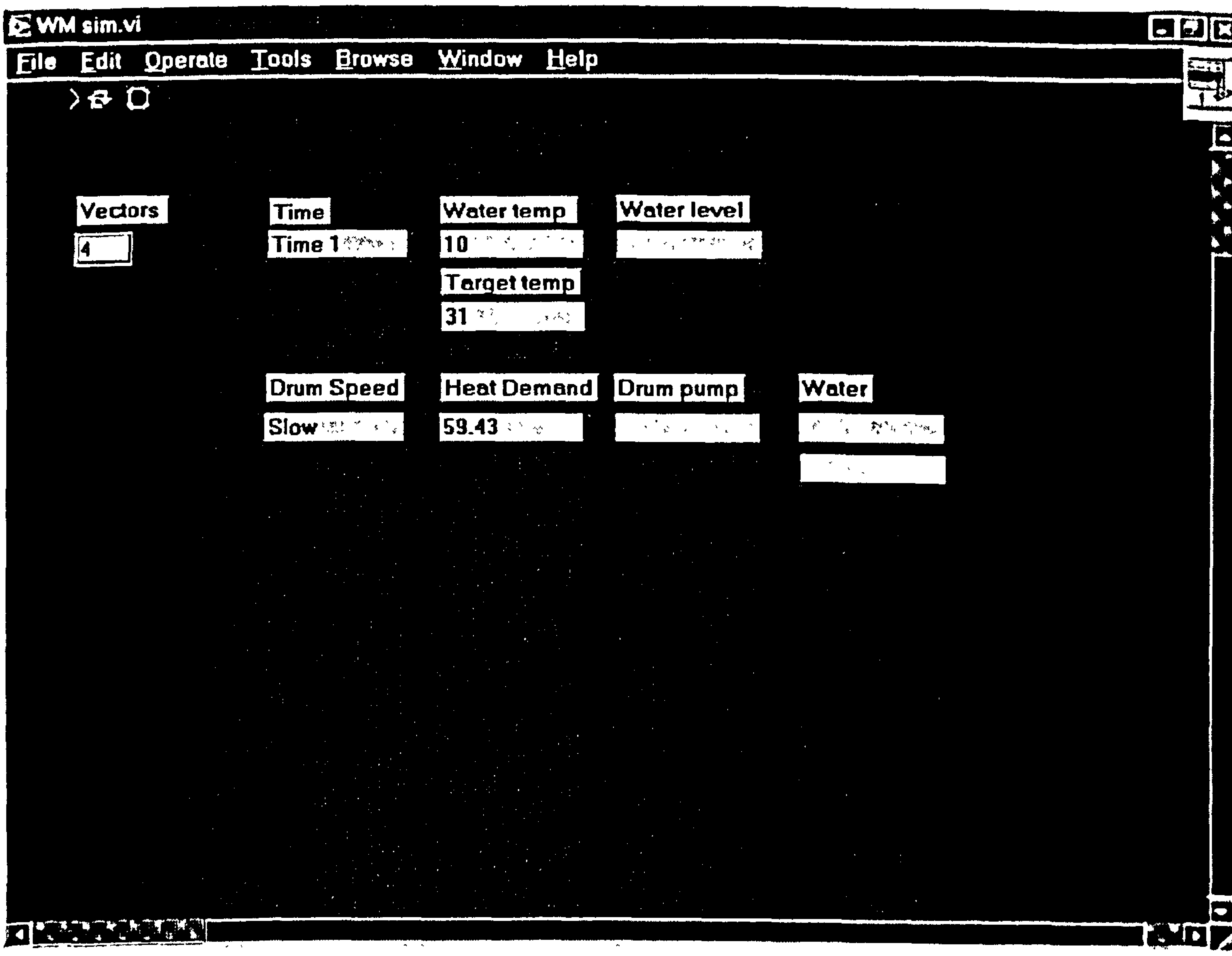

Figure 5.2: Timeslot 1 (Wash). Here the drum rotates slowly whilst the heater demand attempts to correct the water temperature error.

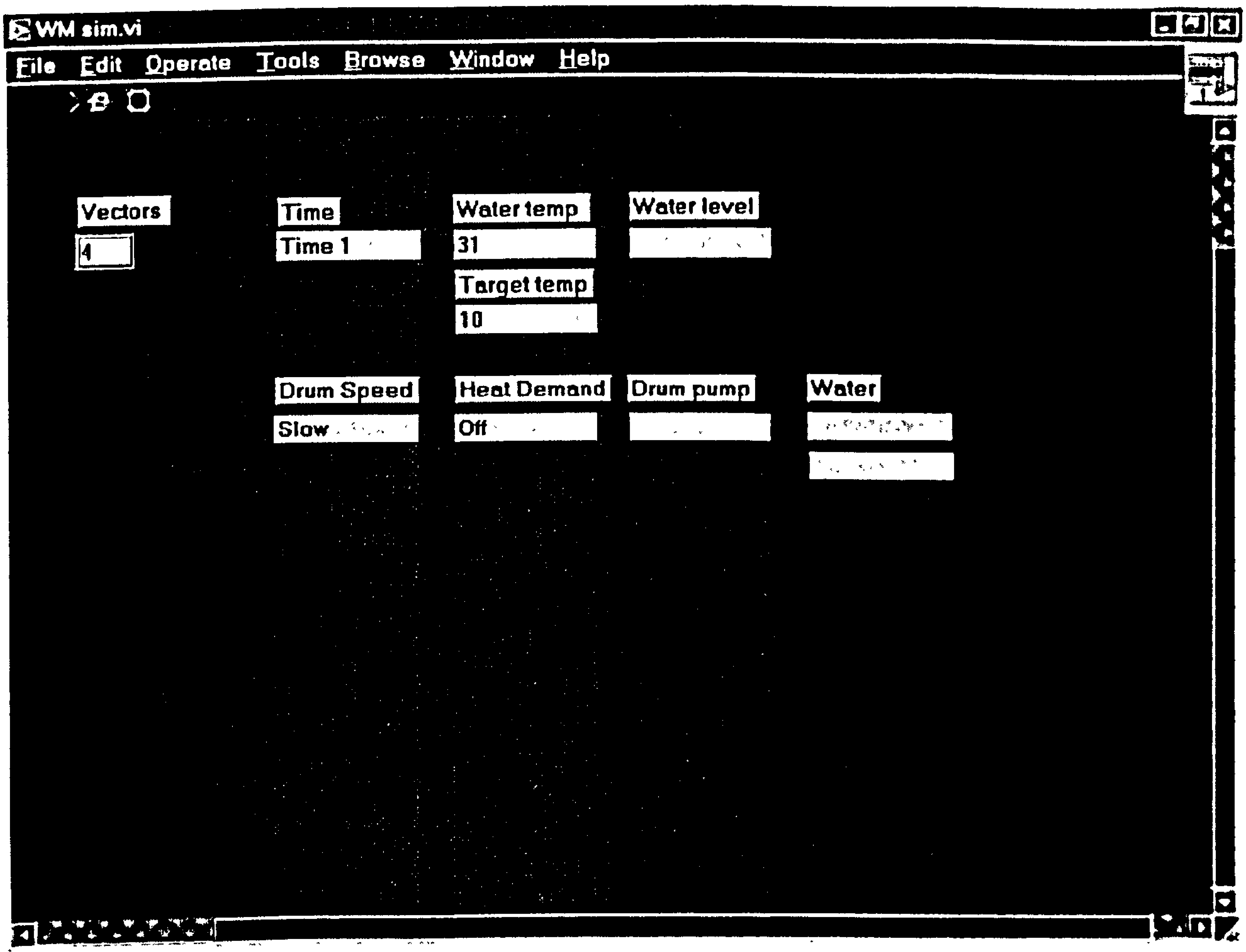

Figure 5.3: Timeslot 1 (Wash). Here the actual water temperature exceeds the desired wash temperature and the Heat logic function has gated out the demand output to prevent a negative demand value being produced.

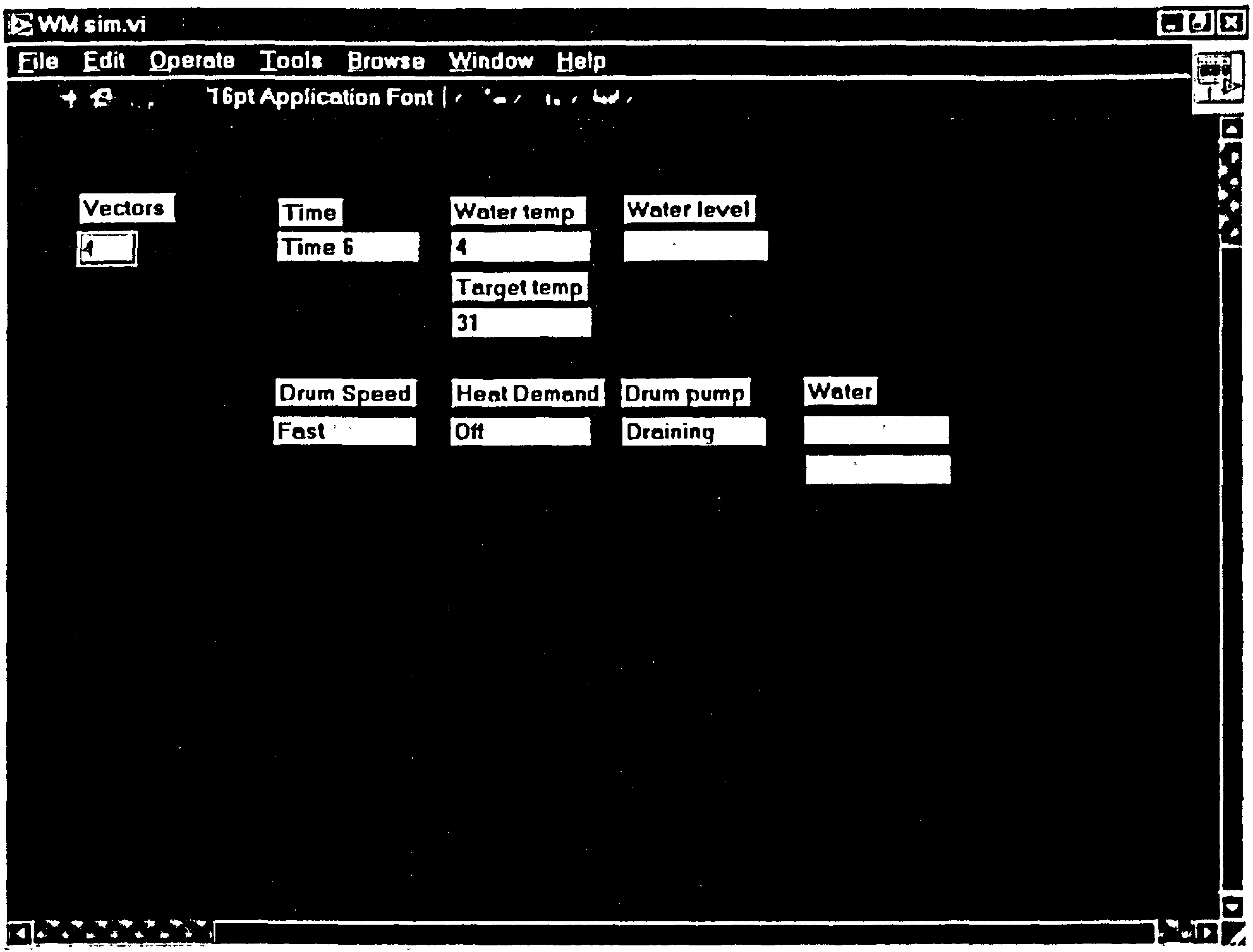

Figure 5.4: Timeslot 6 (Spin). Here the heater is off, the drain pump is active and the drum is on fast rotation.

5.16 Chapter summary

This chapter has covered the problems and solutions found, during the realisation of the technique demonstration outlined in the previous chapter: The results of 900 experimental runs were examined in an attempt to relate search performance, population size, string length and the option of restarting the search: The conclusions drawn were that the LGP system used is very effective – capable of solving the hardest thread here in about half an hour, approximately one in every eighteen times, whilst being relatively insensitive to parameter changes – probably making the technique ideal for wide application: Restarting also proved to be a potentially useful strategy with this problem examined.

Finally, a graphical simulation involving the entire, evolved washing machine operation is described.

In the next chapter, conclusions are drawn about the work carried out here and what has been achieved, and this is followed by suggestions for future work that will carry this research forward.

6 Conclusions

6.1 Introduction

In this final chapter, the work is re - examined and conclusions are drawn about what has been proposed, discovered and completed towards the stated aim of this research; namely, to propose and demonstrate a method of automatic software creation by problem decomposition and evolutionary computing. Later, suggestions are made for Future Work that can add to this research.

6.2 The software creation system developed

The main aim of this research is to propose and demonstrate a method of automatic software creation for certain embedded control problems using concurrent Evolutionary Computing (EC). The target problem class contains non safety – critical problems with a limited requirement for output accuracy, complexity and response timing: Essentially these are problems that can be decomposed into a number of simpler problems that execute in parallel.
The system developed appears to be very effective – capable of producing solutions to the most complex problem thread in approximately half an hour using a three year old PC about one in every eighteen runs: The effectiveness of the search can be appreciated by comparing the frequency and fitness of the best root solutions produced randomly (at best 44 individuals in 20000000 with a fitness of 116.9 [5.15.3]) against the fitness and frequency of the successful runs where a root solution is improved through evolution (attaining a fitness in excess of 75000 in 49 out of 900 runs [tables 5.2 and 5.4]).
A statistical examination of the 900 experimental runs also led to the useful conclusion that, if no fit individuals exist in the population after the first 40% of the total run time allocated, then the computing resources can be better employed by restarting the run with a fresh population rather than continuing [5.15.4].
The relative insensitivity of the system to variations in the population size could be a reassuring indicator of the generic nature of this technique i.e. that the system is robust enough to use on a range of problems without excessive tuning. However the No Free Lunch theorem [*Wolpert & Macready* 1995] suggests that a universal system is unlikely and that some automated

restarting and dynamic parameter tuning may be needed – though this can be reduced if the target problem domain is limited – which is the situation here.

6.3 Problem capture and training set size

Ultimately, the training set necessary would be produced by a front – end tool capable of capturing the problem from the user (as outlined in [3.9]): For both problems here, the training sets were produced manually and, at just sixteen cases per output, are unusually small for EC. This is possible here because the continuous heat function (of the washing machine controller) is linear, and the logic functions trivial: There is no real restriction upon training set size – beyond that of the computing resources available (evolution time will increase in proportion to the time needed to evaluate candidate fitness i.e. present all the training cases). However, the current method employed to assign a (non - zero) fitness score to an individual requires that a nominal level of fitness is exhibited in all training cases [described in 5.4]. This could be a prohibitive requirement for more complex problems described with larger training sets; i.e. this criterion may involve randomly creating a very close solution immediately rather than evolving one. This situation could possibly be tackled by requiring nominal fitness in only a proportion of training cases. Describing more complex functions accurately will always require more samples but such problems are beyond the intended scope of this system i.e. target problems [as detailed in 3.7.2] are those that can be solved by breaking the whole problem into a number of smaller, individual problems that are, in themselves, relatively simple and undemanding in terms of accuracy and reliability.

6.4 Contributions to knowledge: What has been achieved

In summary, the following items have been achieved, and a corresponding contribution to knowledge made in the fields of EC and software engineering in general:

- The examination and classification of Genetic Programming (GP) literature up to mid 2003 – highlighting the lack of work on the decomposition of problems for concurrent EC solutions, or the use of EC to create actual programs for real applications.

- The characterisation of typical embedded control applications and the suggestion that the structural architecture of such software – and the way problems are necessarily decomposed – can prove to be a fruitful EC target. This results in the identification of a realistic and useful, finite application class for GP; and so defines the operational bounds for this technique: This is the class of problems that are non safety – critical, concurrently decomposable into a number of quasi – linear problems, where guarantees about accuracy and response time are not needed [described in 3.7.2].
  Further to this, is the suggestion that GP for concurrent embedded application may benefit from using two distinct data types; Bipolar continuous and Boolean logical.

- The development of a concurrent EC system suitable for use in this research: In addition to implementing a specific language set (created for this work [4.3]) and using two distinct data types (Bipolar continuous and Boolean), successful search required the development and use of the following:

  - A fitness assessment method capable of promoting useful 'root' solutions (i.e. randomly created individuals that exhibit a threshold level of fitness for all training cases applied) and so reduce sub – optimal convergence. (Convergence here is defined as the situation where at least 95% of the population is identical, and that this situation has persisted for at least 50 generations).This method demands that any fitness score attained by a candidate will only be applied if that candidate meets this 'root' solution criterion [5.4].

  - The adoption of a fitness level clamp to damp convergence and so preserve the range of genetic material (population diversity) with the intention of avoiding premature convergence. (Here, premature convergence describes the situation whereby the population has converged upon a solution, but that a better (or optimal) solution is known to exist). The clamp value is a preset maximum value that can be attained by a candidate - irrespective of the degree to which this value is exceeded [5.5].

- A method for evolving solutions for problems involving combined continuous and logical functions: *Here, the functions* are separated during evolution and combined later as a function of the Operating System (OS) [5.14].

- Optimised search with the use of graded mutation to search aggressively and then refine solutions: This is achieved by dividing the overall evolution time allocated into four epochs: During the first epoch, mutation can replace the entire target string; in the next epoch, only one element in that string can be replaced. Mutation in the penultimate epoch may only make coarse adjustment of one constant in the chosen element and, in the final epoch, only fine adjustment of a constant is permitted [5.11]

- The solution of two demonstration control problems repeatedly and quickly using concurrent EC and decomposition: These were a simple fridge controller with two inputs and three outputs, and later a relatively complex washing machine controller with ten inputs and seven outputs.

- An examination into the search sensitivity of the EC system employed here, to changes in population size and program string length [detailed in 5.15.1 and 5.15.2] that shows a useful degree of insensitivity to the search parameters used, thus demonstrating the necessary (relatively) universal nature of this system.
  The same data was used to perform a statistical examination into the merits of early search termination and restarting from zero - if stalled [see 5.15.4]. This yielded a fruitful heuristic for determining the time at which to abandon the current run and so conserve computing resources.

- The development of a simple visualisation tool to demonstrate the operation of the entire control system, and so enhance user confidence [section 5.15.5].

- To identify a number of further research streams that could enhance the technique and the scope of its application [6.6]

6.5 GP and automatic software creation

The past decade of GP research has taken place against a background of exploding computer power, and in excess of 1200 GP papers have been published[11] yet the failure of GP to actually create useful programs is conspicuous: Toy problems, the comparison of improved techniques against benchmarks and various 'curve – fitting' exercises (albeit with possibly complex functions) are no real substitute for the early promise of GP.
One of the earliest claims made for GP by Koza was that GP is capable of discovering not only the solution to a problem, but also the shape of that solution (i.e. architecture, storage etc.) [Koza 1992b]. This may be possible, but is it realistic for anything more complex than trivial problems with a small search space?
This work then, is already a distinct departure from the original GP of Koza because human input is required to define some of the solution shape in advance: This step may be essential though when attempting to solve real problems with software that will reside in chosen, off – the – shelf hardware, or where the dedicated hardware architecture is strictly defined by the problem. In addition, this step could be viewed as a realistic necessity that will actually make larger problems accessible to EC methods.

This work demonstrates how a relatively complex software creation problem – too complex for EC solution as a single program – may become soluble when recast as a number of simpler problems operating in concert.
With regard to scalability; the potential for this technique may be greater than that of tree GP which must identify and capture common modules of functionality automatically in order to tackle larger problems. Here, larger problems can be accommodated by increasing the number of threads – provided this division of overall output is possible.

[11] Online GP bibliography : http://www.ira.uka.de/bibliography/ai/genetic.programming.html

6.6 Future work

This thesis has taken these ideas on automatic program creation using EC in a concurrent processing environment, and extended them from the previous published work [Hart & Shepperd 2002] . Expanding this research further could simply involve bigger or more complex problems; alternatively, these extensions could involve technique refinements or widening the applicability by confidence – building measures and testing:

6.6.1 Towards better performance

Improved search performance, and perhaps a system capable of adapting to a range of problem types automatically, could result from an investigation into dynamic parameter tuning: Failure to converge successfully within a certain number of generations could initiate changes in various evolutionary and system parameters. These changes could be made dynamically and used in the current search, or made prior to a restart.
Suggested parameters could include the fitness threshold required in all training cases [5.4], or in reducing this requirement so that only a proportion of cases need this fitness level. Changes in the mutation operation [5.11] are another possibility, along with the more immediate targets such as mutation and crossover rates, tournament size, initial string length and population size (though sensitivity to population size has proved to be low – for the problems tried [5.15.2]).

The most appropriate representation is vital for success - but this has not been the subject of experimentation here: The idea of using a very simple evolutionary language set may be flawed because the number of instructions needed to be assembled - in the correct order - may make the probability of discovery near zero; whereas a few, more complex instructions, could be more readily assembled. Experiments to determine if the addition of some higher – level instruction would be beneficial – provided that undue customisation of the system to one application area could be avoided.

More direct approaches to deal with the inherent problems of LGP could be addressed by implementing recently published work on linear scaling [Keijzer

2003] that aims to free the LGP to find the desired function - rather than focus on the magnitude of the data.

6.6.2 Widening the range of application

The extension of this work to evolve voting teams [2.7.3.11.2] is an option available that may increase the trustworthiness of the system and so increase the range of possible applications i.e. to include those in which a degree of correctness must be ensured. With voting teams, an output will be formed from the consensus of a number of contributors within a team – thus increasing the reliability of the output generally. This approach to safety – critical software development by humans has already been investigated in the form of N version software development [Hatton 1997]. GP forms of N version software have been explored in [Soule 1999] and [Imamura & Foster 2001], and similar approaches to forcibly create heterogeneous team members (by manipulating the evolutionary context) in [Feldt 1998].

Further confidence in the operation of the evolved controller may result from an investigation into the use of run – time bounds checking[12] and watchdog timers[13] – already a common feature in embedded control systems.

6.6.3 Testing

Testing, and design – proving in general, will usually be an essential element in the automatic creation of software but, beyond the graphical demonstration system developed [5.15.5] this work has not been attempted here: There remains great scope for work utilising *automated test vector generation* to accompany this work. The effect of intensive automated testing could be to raise confidence in the technique and so increase the range of target applications.

Alternatively, because of the fast evolution time, and consequent low cost of redesign, the whole system could be used (along with the front end tool [3.9]) to provide a cyclic prototype – redesign – trial approach to development - with the system user / product designer and target hardware: This type of testing

---

[12] Run time bounds checking:
This is a feature available with languages targeted at safety – critical applications such as ADA [Barnes 1989]. Here, the normal bounds for a program variable or output are specified prior to execution. If the variable exceeds these bounds, exception processing is initiated.

[13] Watchdog timers:

can, in many cases, be viewed as the ultimate test because it does not use any abstraction away from the target.

6.6.4 Front end tool

Beyond stating the output requirements of this tool for problem capture [3.9] no work has been attempted on its development. The generation of automatic test vectors could be an additional output of such a tool – though these would only test against the captured model of the required software and not necessarily the required software.

Watchdog timers are typically hardware counters that will reset the microprocessor upon expiration – returning the 'processor from a known start, state. During normal execution of the control operation, the microprocessor must regularly reload the watchdog timer to prevent the fault restart.

References

Acarnley & Al-Sadiq 2002:

Acarnley, P. & Al-Sadiq, Y, Tuning PI speed controllers for electric drives using simulated annealing, ISIE 2002. Proceedings of the 2002 IEEE International Symposium on Industrial Electronics Cat. No.02TH8608C., 2002: 1131-5 vol.4; IEEE, Piscataway, NJ, USA

Ahluwalia & Bull 1998:

Ahluwalia M, Bull L, Co-evolving functions in genetic programming : Dynamic ADF creation using GliB Lecture notes in computer science, 1998, Vol.1447, pp.809-818 CO: Evolutionary programming. International conference (7), San
Diego CA, 1998-03-25

Albuquerque et al 2000:

Albuquerque, P., b. Chopard, et al. (2000). On the impact of the representation on fitness landscapes. *Euro GP 2000, Springer-Verlag* LNCS.

Andre & Teller 1999:

Andre D, Teller A. Evolving team darwin united, Lecture notes in computer science, 1999, Vol.1604, pp. 346-351

Angeline 1997:

Angeline PJ, The benefits of distributed solutions when evolving symbolic equations International Society for Optical Engineering, Bellingham WA, United States SPIE proceedings series, 1997, Vol.3165, pp.124-134

Banzhaf et al 1998a:

Banzhaf W,, Nordin P., Keller R., Francone F., Genetic Programming An Introduction. Morgan Kaufmann Publishers Inc. 1998.

Banzhaf et al 1998b:

Banzhaf W,, Nordin P., Keller R., Francone F., Genetic Programming An Introduction, page 132. Morgan Kaufmann Publishers Inc. 1998.

Banzhaf et al 1998c:

Banzhaf W,, Nordin P., Keller R., Francone F., Genetic Programming An Introduction, page 96. Morgan Kaufmann Publishers Inc. 1998.

Barnes 1989:

Barnes JGP., Programming In Ada. 1989. Addison Wesley Publishing Ltd.

Beasley et al 1993a:

Beasley, D., D. R. Bull, et al. (1993). An Overview of Genetic Algorithms: Part 1, Fundamentals. University Computing 15(2): 58-69.

Beasley et al 1993b:

Beasley, D., D. R. Bull, et al. (1993). An Overview of Genetic Algorithms: Part 2, Research Topics. University Computing 15(4): 170-181.

Bennett et al 1999:

Bennett F. H., Koza J., Keane M. Andre D., Darwinian programming and engineering design using genetic programming. Proceedings of the first international workshop on soft computing applied to software engineering. 1999, pp 31-40. Limerick university press.

Bongard 2000:

Bongard JC., The Legion system: A novel approach to evolving heterogeneity for collective problem solving. 2000. Proceedings of EuroGP 2000 : European conference on genetic programming, Edinburgh, vol. 1802, pp. 16-28. Springer – Verlag, Berlin.

Bremermann 1962:

Bremermann H., Optimization through evolution and recombination. In Yovits M, Jacobi G, Goldstein G (editors) Self organising systems pp 93-106 Spartan books, New York.

Bunyaratavej & Miller 2002:

Bunyaratavej, P.; Miller, D. J. An iterative hillclimbing algorithm for discrete optimisation on images: application to joint encoding of image transform coefficients. IEEE Signal Processing Letters. Feb. 2002; 9(2): 46-50.

Cantu - Paz 2002:

Cantu-Paz E., On random numbers and the performance of genetic algorithms. Proceedings of Genetic and Evolutionary Computing Conference (GECCO) New York 2002, Morgan Kaufmann Publishers.

Clark et al 2003:

Clark, J., J. J. Dolado, et al. (2003). Reformulating software engineering as a search problem. IEE Proceedings Software 150(3): 161-175.

Ebner 1999:

Ebner M., Evolving an environment model for robot localisation, Lecture notes in computer science, 1999, Vol.1598, pp.184-192

Feldt 1998:

Feldt R, Generating diverse software versions with genetic programming: an experimental study IEE Proceedings Software. vol.145, no.6; Dec. 1998, p.228-3

Fernandez et al 2000:

Fernandez F, Tomassini M, Punch WF, Sanchez JM, Experimental study of multipopulation parallel genetic Programming, Lecture notes in computer science, 2000, Vol.1802, pp.283-293

FigueiraPujol & Poli 1998:

FigueiraPujol JC, Poli R., Efficient evolution of asymmetric recurrent neural networks using a PDGP inspired two dimensional representation, Lecture notes in computer science, 1998, Vol.1391, pp.130-141

Fogel et al 1965:

Fogel L, Owens A, Walsh M. Artificial intelligence through a simulation of evolution. In Maxfeild M, Callahan A, Fogel L, editors. Biophysics and Cybernetics Systems pages 131-155.

Friedberg 1958:

Friedberg R., 1958, A learning machine. Part 1 IBM J. Research and Development 2:2-13.

Fuchs 1999:

Fuchs M, Large populations are not always the best choice in genetic programming, GECCO-99. Proceedings of the Genetic and Evolutionary Computation Conference. 1999: 1033-8 vol.2, Morgan Kaufmann Publishers, San Francisco, CA, USA

Glover 1990:

Glover F., Tabu Search: A tutorial. Interfaces 20 (1990) pp. 74 – 94

Goldberg 1989:

Goldberg D., Genetic algorithms in search, optimisation and machine learning 1989. Addison-Wesley, Reading MA.

Gross 2000:

Gross H G., Measuring evolutionary testability of real-time software. (PhD thesis) 2000. The British Library shelf mark DXN 044 3700.

Hatton 1997:

Hatton L, N version design versus one good version, IEEE Software, 1997, vol 14, part 6, pp. 71-76.

Hart & Shepperd 2002:

Hart J., Shepperd M., Evolving software with multiple outputs and multiple populations. Late breaking Papers, Proceedings of Genetic and Evolutionary Computing Conference (GECCO) New York 2002, Morgan Kaufmann Publishers.

Haynes 1998:

Haynes T, Random search versus genetic programming as engines for collective adaptation. Lecture notes in computer science, 1998, Vol.1447, pp.683-692 CO: Evolutionary programming. International conference (7), San
Diego CA, 1998-03-25

Holland 1975:

Holland J.,1975 Adaptation in natural and artificial systems. MIT Press, Cambridge MA

Howard et al 1999:

Howard D, Roberts SC, BrankinR, Target detection in SAR imagery by genetic programming Advances in Engineering Software, 1999, Vol.30, No.5, pp.303-
311

Imamura & Foster 2001:

Imamura K, Foster J. Fault tolerant computing with N version genetic programming. GECCO 01. Proceedings of the Genetic and Evolutionary Computation Conference, p178, 2001. Morgan Kaufmann Publishers.

Keijzer et al 2001:

Keijzer M., Ryan C., O'Neill M., Cattolico M., Babovic V., Ripple crossover in genetic programming, Proceedings of the 4th annual conference on genetic programming, EuroGP Lake ~Como, Italy 2001, Springer-Verlag Berlin Heidelberg 2001

Keijzer 2003:

Keijzer M., 2003, Improving Symbolic Regression with Interval arithmetic and Linear Scaling, EuroGP2003 6th European conference proceedings. Springer-verlag Berlin Heidelberg 2003

Knuth 1997:

Knuth D., The art of computer programming Vol 2. 1997 page 104 Addison – Wesley.

Koza 1992a:

Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. 1992, Cambridge, MA: MIT Press.

Koza 1992b:

Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. Chapter 1: Introduction.1992, Cambridge, MA: MIT Press.

Koza 1992c:

Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. Part 3: Architecture altering operations: Chapter 3.7. 1992, Cambridge, MA: MIT Press.

Koza 1992d:

Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. Chapter 9: Automatically defined storage. 1992, Cambridge, MA: MIT Press.

Koza 1992e:

Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. Chapter 57: Evolvable hardware and rapidly reconfigurable Field Programmable Gate Arrays. 1992, Cambridge, MA: MIT Press.

Koza et al 2000:

Koza J R, Bennett F H, Andre D, Keane M A, Synthesis of topology and sizing of analog electrical circuits by means of genetic programming, Computer methods in applied mechanics and engineering, 2000, Vol.186, No.2-4, pp.459-482

Langdon 2000:

Langdon W B. Size fair and homologous tree genetic programming crossovers. Genetic Programming And Evolvable Machines, vol. 1 1, pp. 95-l 19, Apr. 2000.

Langdon & Poli 2002:

Langdon W. B. & Poli R. Foundations of Genetic Programming. Springer-Verlag Berlin and Heidelberg, 2002

Langdon 2003:

Langdon W B, Convergence of Program Fitness Landscapes, Genetic and Evolutionary Computing Conference GECCO 2003, Chicago, Morgan Kaufmann Publishers

Leung & Wong 1995:

Leung, K., Wong M., Applying logic grammars to induce subfunctions in genetic programming. In Proceedings of the 1995 IEEE Conference on Evolutionary Computation, pages 737- 740. USA:IEEE Press.

Leventhal, 1978:

Leventhal L. R. An introduction to 6800 assembly language programming. 1978. Osborne & Associates Inc., Berkeley, California.

Lyu 1993:

Lyu M, Experience in Metrics and Measurement in N Version programming,
International Journal of Reliability, Quality and Safety Engineering, 1993,1, (1), pp 41-62,

Martin et al 1998:

Martin L, Moal F, Vrain C. A relational Data Mining tool based on Genetic Programming, Lecture notes in computer science, 1998, Vol.1510, pp.130-138

Martin 2000:

Martin P, Genetic programming for service creation in intelligent networks. Lecture notes in computer science, 2000, Vol.1802, pp.106-120 EuroGP 2000 : European conference on genetic programming, Edinburgh, 200004-15

Menon 2002:

Menon A., The point of point crossover: Shuffling to randomness Proceedings of Genetic and Evolutionary Computing Conference (GECCO) New York 2002, Morgan Kaufmann Publishers.

Meysenburg & Foster 1999:

Meysenburg M M, Foster J A, Randomness and GA performance, revisitedGECCO-99. Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers, San Francisco, CA, USA; 1999; 2 vol. xvi+1876 pp. .425-32 vol.1

Meysenburg et al 2002:

Meysenburg M., Hoelting D, McElvain D, Foster J, How random generator quality impacts genetic algorithm performance. Proceedings of Genetic and evolutionary Computing Conference (GECCO) New York 2002, Morgan Kaufmann Publishers.

Nikolaev & Iba 1999:

Nikolaev N, Iba H., Automated discovery of polynomials by inductive genetic programming. Lecture notes in computer science, 1999, Vol.1704, pp.456-461

Nordin 1999:

Nordin P, Eriksson A, Nordahl M. Genetic reasoning : Evolutionary induction of mathematical proofs. Lecture notes in computer science, 1999, Vol.1598, pp.221-231 *EuroGP 99: European workshop on genetic programming*

Nordin et al 1999:

Nordin P., Banzhaf W., Francone F., Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. Advances in Genetic Programming 3, pp. 275 – 299. 1999, MIT press.

Nyongesa 2001:

Nyongesa H O., Kent S., R. O'Keefe. Genetic *programming for anti – air* missile proximity fuse delay time algorithms. IEEE Aerospace and Electronics systems magazine. 2001, vol. 16, part 1, pp 41-45.

O'Neill & Ryan 1999:

O'Neill M, Ryan C. Evolving multi-line compilable C programs Lecture notes in computer science, 1999, Vol.1598, pp.83-92

Ostrowski & Reynolds 1998:

Ostrowski D A, Reynolds R G, Integration of slicing methods into a cultural algorithm in order to assist In large scale Engineering systems design. Lecture notes in computer science, 1998, Vol.1447, pp.191-198. Evolutionary programming. International conference, San Diego CA, 1998-03-25

Pelletier & Weinerskirch 2002:

Pelletier O, Weinerskirch A, Algorithmic self assembly of DNA tiles and its application to cryptoanalysis. Proceedings of Genetic and Evolutionary Computing Conference (GECCO) New York 2002, Morgan Kaufmann Publishers.

Podgorelec & Kokol 2000:

Podgorelec V, Kokol P,Fighting program bloat with the fractal complexity measure. Lecture notes in computer science, 2000, Vol.1802, pp.326-337. EuroGP 2000 : European conference on genetic programming, Edinburgh, 2000-04-15

Poli & Langdon 1997:

Poli R, Langdon W B. Genetic programming with one point crossover. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, Soft Computing in Engineering Design and Manufacturing, pages 180—189. Springer-Verlag London, 1997.

Poli 2001:

Poli R, General Schema Theory for GP with subtree swapping crossover. Euro GP 2001 proceedings. Springer – Verlag.

Popp et al 1998:

Popp R L; Montana D J; Gassner R R; Vidaver G; Iyer S, Automated hardware design using genetic programming, VHDL, and FPGAs. SMC 98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.98CH36218). IEEE, New York, NY, USA; 1998; 5 vol. 4945 pp. 2184-9

Poli 1999:

Poli R, Sub-machine code GP: New results and extensions Lecture notes on Computer *Science, 1999, Vol.*1598, pp.65-82

Rechenberg 1973:

Rechenberg I., Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. Frommann – Holzboog Verlag Stuttgart 1973

Reetinder et al 1999:

Reetinder P. Sidhu, Alessandro Mei, and Viktor K. Prasanna, Genetic Programming using Self Reconfigurable FPGAs, International Workshop on Field Programmable Logic and Applications, Sep. 1999. Prashanth

Ryan et al 1998:

Ryan C; Collins J J; O'Neill M, Grammatical Evolution: evolving programs for an arbitrary language. Genetic Programming. First European Workshop, EuroGP 98. Proceedings. Springer-Verlag, Berlin, Germany; 1998; x+232 pp.p.83-96, 1998

Ryan 1999:

Ryan C., Automatic Reengineering of Software Using Genetic Programming 1999 Kluwer.

Ryan & Ivan 2000:

Ryan C, Ivan L, Paragen : The first results Lecture notes in computer science, 2000, Vol.1802, pp.338-348 CO: EuroGP 2000 : European conference on genetic programming, Edinburgh, 2000-04-15

Savic et al 1999:

Savic D A, Walters G A, Davidson J W., A genetic programming approach to rainfall runoff modelling, Water resources management, 1999, Vol.13, No.3, pp.219-231

Sidhu et al 1999:

Sidhu R P S, Mei A, Prasanna V K, Genetic programming using self reconfigurable FPGAs, Lecture Notes in Computer Science, 1999, Vol.1673, pp.301-312

Soule 1999:

Soule, T. (1999). Voting teams: a cooperative approach to non-typical problems using genetic programming. GECCO-99. Proceedings of the Genetic and Evolutionary Computation Conference (GP-99). Morgan Kaufmann Publishers.

Spector et al 1998:

Spector L, Barnum H, Bernstein H, Genetic programming for quantum computers. In Genetic Programming 1998, Proceedings of the third annual conference pp. 365-374 Morgan Kaufmann.

Spector 2002:

Spector L, Quantum computing for Genetic programmers, Tutorial Program, Genetic and Evolutionary Computing Conference (GECCO) New York 2002,

Stallings 1990:

Stallings W., Reduced Instruction Set Computers. 1990. IEEE Computer Society Press. Los Alamos, California.

Vallejo & Ramos 2000:

Vallejo E E, Ramos F, Evolving insect locomotion using cooperative genetic programming Lecture notes in computer science, 2000, Vol.1793, pp.170-181 MICAI 2000 : Mexican international conference on artificial intelligence, Acapulco, 2000-04-11

Watts 1998:

Watts J M, Animats : Computer simulated animals in behavioral research, Journal of animal science, 1998, Vol.76, No.10, pp.2596-2604

Whigham 1995:

Whigham P. 1995. Grammatically-based Genetic Programming. In Proceedings of the Workshop on Genetic Programming: From Theory to Real. World Applications, pages 3-41. Morgan Kaufmann Pub.

Whitley 2001:

Whitley D., An overview of evolutionary algorithms: Practical issues and common pitfalls. Information and software technology, 2001, vol. 43, pp. 817-831.

Williams & Williams 1999:

Williams K P; Williams S A, Two evolutionary representations for automatic parallelization. GECCO-99. Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers, San Francisco, CA, USA; 1999; 2 vol. xvi+1876 pp.1429-36

Wolpert & Macready 1995:

Wolpert D, Macready W, No free lunch theorems for search, SFI-TR-95-02-010 The Sante Fe Institute.

Yeun et al 2000:

Yeun Y S, Suh J C, Yang Y S, Function approximations by superimposing genetic programming trees: with applications to engineering problems INFORMATION SCIENCES, 2000, Vol.122, No.2-4, pp.259-280

Zaks & Lesea, 1979:

Zaks R., Lesea A., Microprocessor interfacing techniques. Third Edition Published 1979, Sybex Inc.

Zhang & Cho 1999:

Zhang B T, Cho D Y., Genetic programming with active data selection, Lecture notes in computer science, 1999, Vol.1585, pp.146-153

Appendix A

Evolutionary language set

Appendix A: Evolutionary Language Grammar.

The evolutionary language set used is { N, T, P, S }

where S is the start symbol, T is the set of terminals, N the set of non-terminals and P the set of production rules that map N to T.

Terminals - T

These are items that can appear in the language.

T = { +, -, *, /, >, <, =, IF, AND, OR, XOR, NOT, (,), <branch_number>, <line_number>, THEN, <const> }

Non Terminals - N

These can be broken down into one or more Terminals and non-Terminals

N = { code, line, expr, if_op, op, bool_op, cont_op, comp_op }

Production rules - P

(1) <code> :: = <line> (A)

|<code><line> (B)

(2) <line> :: = <expr>

(3) <expr> ::= <if_op> (A)

| <Out_Item>=<op> (B)

(4) <op> :: = <In_item1><cont_op><In_item2> (A)

|<In_item1><comp_op><In_item2> (B)

|<In_item1><bool_op><In_item2> (C)

| NOT <In_Item2> (D)

(5) <Out_Item>::= { Me.<nm>, Ou.<no> }

(6) <In_Item1>::= { Me.<nm>, In.<ni>, Const }

(7) <In_Item2>::= { Me.<nm>, In.<ni>, <Const> }

(8) <Const>::= Constant floating point value in the range -10.0 to 10.0

(9) <cont_op>::= { +, -, *, / }

(10) <comp_op>::= { <, > }

(11) <bool_op>::= { AND, OR, XOR, NOT }

(12) <if_op>: = IF (<In_Item1>) THEN (<line number >)

(13) <nm>::= { 0, 1 }

(14) <ni>::= { 0,1 }

(15) <no>::= { 0, 1 }

(16) <line number>::= { (Current line number + 1,...,Last possible line number) }

where:

The above describes a two input, two memory, two output example configuration.

Continuous operations (4.A) interpret operands as floating point, bipolar values and produce similar output.

Comparison operations (4.B) compare operands that are assumed to be bipolar, *floating* point values but produce a boolean result of 1 (1.0) if the operation evaluates to TRUE, and 0 otherwise.

Boolean operations (4.C) consider operands to be FALSE if zero and TRUE otherwise. Operations are logical in nature and the result wil be 1 if the operation evaluates as TRUE, and 0 otherwise.

The 'if' operation (12) will evaluate <In_item1> as FALSE if this item is zero, and TRUE otherwise. With a TRUE result, the next line executed will be that specified by <line_number> and all intermediate lines (if any) will be ignored and consequently any outputs specified in these lines will not be updated.

Appendix B

Gaussian random number generation

Appendix B: Gaussian random number generation

The Microsoft Visual C++ compiler RAND( ) function produces only uniformly distributed pseudo random numbers. To generate the normal gaussian distributed random numbers needed for the mutation operation, the Polar Method described in 'The Art of Computing' [Knuth 1997] was encoded:

To generate two normally (gaussian distributed) distributions from a uniformly distributed random number sequence:

Take two random numbers between 0 and 1, u1 and u2.

Shift these to make two numbers that lie between +1 and -1, v1 and v2 by;

$$v1 = 2u1 - 1 \text{ and } v2 = 2u2 - 1$$

Calculate $S = v1^2 + v2^2$

If S >= 1 then repeat the above steps, otherwise:

Two normally distributed random numbers can be found by:

$$x1 = v1 \text{ sqr } (-2 \ln S / S)$$
$$x2 = v2 \text{ sqr } (-2 \ln s / s)$$

where sqr( ) returns the square root and ln( ) returns the natural logarithm.

Appendix C

Fitness of random strings at creation

Appendix C: Fitness of random strings at creation

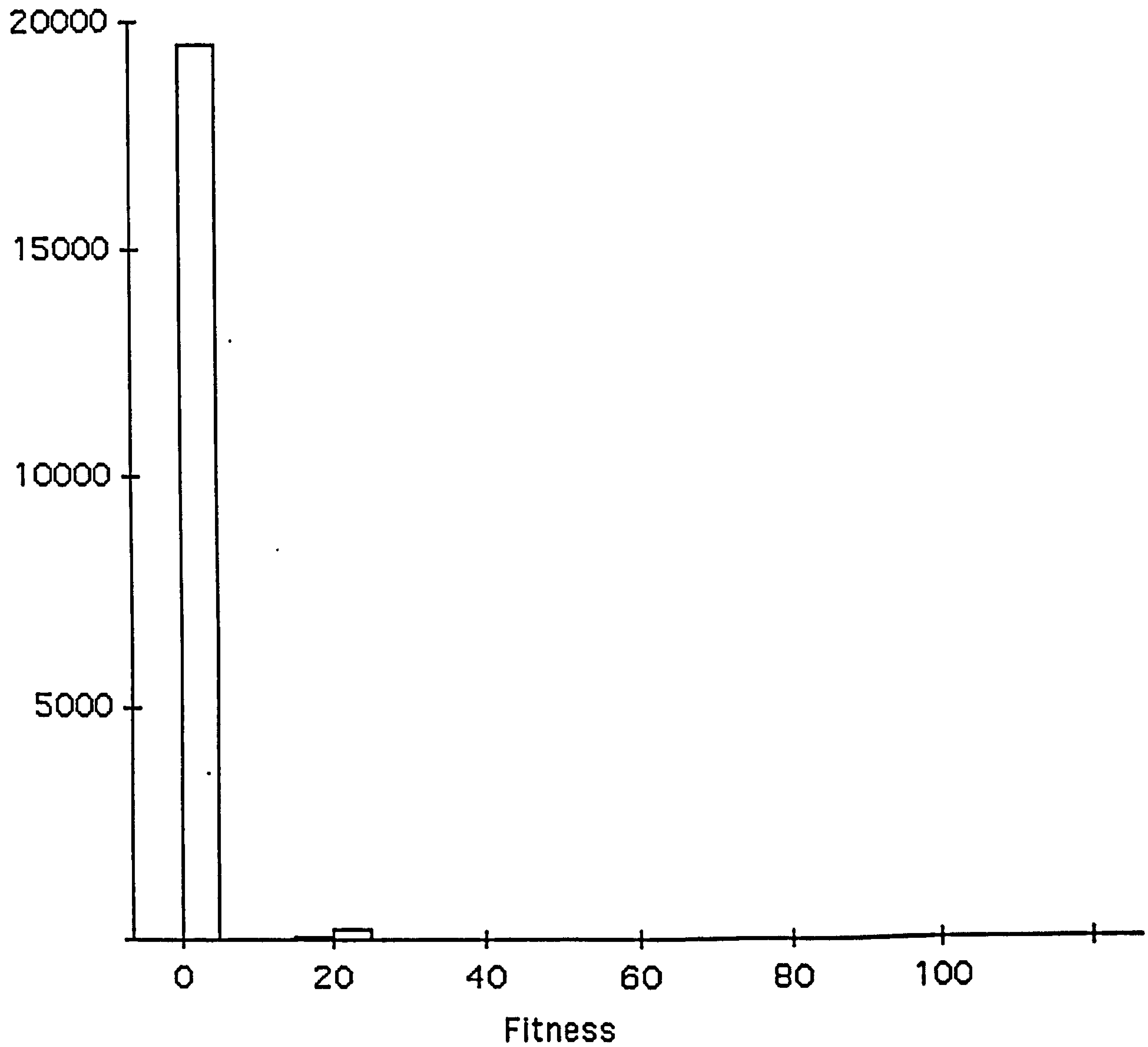

Figure C.1: Histogram of random strings at creation

Fitness Range	Frequency
0-4	19511
5-9	0
10-14	10
15-19	87
20-24	251
25-29	42
30-34	0
35-39	11
40-44	11
45-49	22
50-54	0
55-59	0
60-64	0
65-69	0
70-74	0
75-79	0
80-84	0
85-89	0
90-94	0
95-99	0
100-104	0
105-109	0
110-114	11
115-119	44

Table C.1: Fitness distribution of random string individuals at creation.