

The Evolution of Concurrent Control Software Using Genetic Programming

John Hart and Martin Shepperd

Bournemouth University
jhart, mshepper@bournemouth.ac.uk

Abstract. Despite considerable progress in GP over the past 10 years, there are many outstanding challenges that need to be addressed before it will be widely deployed for developing useful software. In this paper we suggest a method for the automatic creation of concurrent control software using Linear Genetic Programming (LGP) and a ‘divide and conquer’ approach. The method involves decomposing the whole problem into a multi-task solution with multiple inputs and multiple outputs – similar to the process used to implement embedded control solutions. We describe the necessary architecture of typical embedded control systems and their relevance to this work, the software evolution scheme used and lastly demonstrate the technique for an embedded software problem, namely a washing machine controller.

Keywords: linear genetic programming, embedded software.

1 Introduction

Genetic Programming [5] has been a research interest since the early 1990s, and can be viewed as an extension of Genetic Algorithm research [4]. Both techniques have many similarities with the earlier work on Evolutionary Strategies [9] and even the first Evolutionary Computing (EC) work of Friedberg in the 1950s [2].

Despite considerable research effort globally towards refining the existing GP techniques, and the development of variants like Grammatical Evolution [10], the ability to create useful, complex and problem-specific programs has remained elusive. Perhaps this is not surprising given that creating computer programs by GP can be seen as a Markov process¹ and, as such, the longer the program is (and so the greater the complexity) then the probability of its discovery will decrease exponentially. The need to search for simpler programs has been identified by other research groups, e.g. Leung *et al.* [8] in their work evolving parallel code.

In this paper we note that the necessary steps required to implement embedded control solutions effectively divide the overall problem into a number of

¹ 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. This model has been used to construct analyses of GP operation [7].

simpler problems. We argue that a similar problem decomposition process can enable EC to automatically generate useful programs for a certain class of problems. This approach is a departure from the original aspirations for GP made by Koza where he suggested that GP can find not only the solution but also the shape of that solution. Perhaps this is unrealistic for anything beyond trivial problems?

Instead we propose a system that requires the problem shape to be defined in terms of the required program outputs. Thus the target user group for this method of software evolution would include product designers / engineers who wish to add embedded control to their product but who don't have the time, money or interest to write their own software. Instead the contribution made by these users would be the definition of embedded control system inputs and outputs, and the required relationship between these items in operation.

The method described here will only be applicable to a certain class of problems. Suitable targets may be defined as non safety-critical, not demanding high precision or guaranteed response timing or reliability. All such targets would still need to be relatively simple, or capable of being decomposed into a solution with such a profile. Hence suitable applications could include fridges, TV remote controls, building environment management systems, dishwashers, etc.

The rest of this paper is organised as follows: Section 2 describes typical characteristics of embedded control systems and how these are relevant; Section 3 outlines the problem decomposition and the EC system used to evolve the solution, Section 4 describes a washing machine controller as an example problem. Section 5 describes the results obtained from tackling this problem, and finally Section 6 draws conclusions from this research, the example problem and ideas for future research.

2 Embedded control systems

An embedded control system will typically comprise of dedicated hardware hosting dedicated embedded software or firmware (i.e. between hardware and software). Usually the hardware and software will be optimised or minimised solely to perform the task at hand; there is not usually a requirement for flexibility or expansion. Many embedded applications must guarantee response and execution timing which is possible since the application does not share the hardware and because execution prioritisation is achieved through the use of exception processing i.e. in response to timer or external events. Here, execution will transfer to the code necessary to respond to that event (unless interrupted again by a higher priority event). In effect, structuring the code in this way, segments much of the entire application into a number of execution 'channels' initiated by an event, and linking input to output via the channel code.

Another distinct feature of many embedded control applications is the use of multiple inputs and outputs, for example, a robot in a car plant may have inputs for object position, belt speed, object orientation in two dimensions etc., and outputs for tool position in three different planes etc. This is in contrast to

PC type applications that might typically only deal with keyboard input and screen output.

Indirectly then, this separation of overall application output into a number of separate outputs, and the demand for temporal as well as structural hierarchies in the code to guarantee response timing, have the effect of decomposing the entire problem into a number of smaller problems, each using a separate channel through execution time and space.

These considerations lead to the approach adopted in this work whereby the overall problem is viewed as a number of smaller code segments each feeding a separate output, such that each isolated code channel is simple enough to evolve automatically using EC techniques.

3 Automatic software evolution method

We obtain a problem decomposition by evolving separate GP code for each channel so that each channel will contribute data for one system output. The channel code is evolved using an isolated population and ultimately the evolved code is intended to be run as a separate concurrent task and synchronised by a multi-tasking OS or implemented on a multi-processor platform.

The goal of this research was to create and evaluate a general purpose software evolution method that could be applied to a range of applications without specific tuning. Consequently relatively little effort has been expended in optimising the performance of the evolutionary method or parameters used, although it turned out that system performance is relatively insensitive to different parameter values within the ranges we studied (see Section 5).

A form of Linear Genetic Programming [1] was chosen because of the relative ease of implementation and inherent problems such as bloat² are reduced. Consistent with the desire to build a system with general applicability, the evolutionary language set used was kept simple and general. Effectively, this could be seen as a RISC-type³ approach, which led to very efficient use of computing

² Program bloat is mainly a phenomenon manifested in tree-based GP whereby crossover can create trees of excessive size, through the combination of large subtrees. This will often result in an excessive and possibly inefficient use of computing resources. The bloat, however, can be seen as important in evolutionary terms because the introns (unused code) – that can form the bulk of the bloat – will serve to protect the executed code from mutation. Bloat is often tackled by combining a fitness award for brevity with the existing fitness function.

³ RISC processors use what is perhaps a counterintuitive 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 becomes simpler. This simplification allows the processor to run faster such that even with a greater number of instructions to process, the overall execution time will usually be reduced.

resources i.e. simple instructions that can be compiled to a single machine code instruction.

As a variant to the standard LGP where execution flow is linear and contiguous, we have implemented a branch instruction to allow the formation of iteration in the evolved code strings if required. The functions available for each language statement are the mathematical operators (+, -, *, /⁴) the logical operators (AND, OR, XOR, NOT) and the relational operators (<, >). In addition, a conditional branch instruction is included.

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'. Here, the same storage class is used for both types 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, thus, zero is interpreted as FALSE 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 relational operators 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 can be used for reading numeric values from various sources e.g. to sample a continuous value such as temperature, position or time. The Boolean type input / output can be used for logic level type sensing e.g. control switches, limit sensors, etc.

The LGP program strings are of variable length (up to a preset maximum) and consist of atomic statements. With the exception of the IF branch instruction, each statement has 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 direct flow and construction of information up through the program i.e. the separate memory registers available are indexed.

Beyond the string length and population size variations, the mechanisms of EC used were basic and were not investigated with a view to optimisation. 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,

⁴ Note that the divide operator is protected by returning 1 upon attempts to divide by zero.

for each population (task) in case the ability to adjust the evolution process for each output / function became necessary.

To avoid a possibly unhelpful saturation of the population by a single candidate, a maximum fitness ceiling value was employed. This clamp value was used as a candidates 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.

One immediate side effect of clamping the maximum fitness score in this way, was 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 could be seen as acceptable because of the stated problem class targeted here.

Demanding that any solution to be classed as a 'root' solution exhibits threshold fitness in all training vector cases applied created 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 was achieved simply multiplying the fitness score by a constant.

Crossover action on LGP systems had limited impact. It made little difference (in most cases) in which order unrelated program statements were executed, thus, the action of exchanging the top and bottom halves of two strings, in a converged population contributed little. However, this ordering was certainly more critical in the case of the Output statement. In this case, the sequential execution of the program strings (unless branching occurs) created a situation whereby only the final Output statement executed (and the code that built the data for this) was important. Perhaps not unexpectedly then, the level of crossover used had little impact on the level of success attained. The crossover rate used was 25%.

Subsequent to the initial population, the only way of introducing diversity into the population was 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 investigated, along with the replacement of the least fit individuals in the population with an equal number of randomly created strings. The most effective method used in the experiments was a hybrid approach. The maximum evolutionary time allowed is 20000 generations (interrupted upon success) and this time was divided into four epochs, with the level or type of mutation applied changed according to the current epoch:

- Epoch 0 (the first 40% of evolution time): one single statement is chosen randomly from the string selected for mutation and the entire statement replaced.

- Epoch 1 (40% to 70% of evolutionary time): only one part of the selected statement is altered (either the operation type, the output channel type or one of the two input channels).
- Epoch 2 (70 to 90% of evolution time): only input constants are modified in the chosen statement – if used, the current constant value is retained but adjusted with the addition of a bipolar random number chosen from a Gaussian distribution with a mean of 0, and a standard deviation of 2.
- Epoch 3 (final 10% of evolutionary time): as per epoch 2, but using a standard deviation of 1.

The intention of this approach was to focus down on the solution by using less aggressive mutation over time and concentrating any adaptation only on constants towards the end. Thus, by incrementally adapting the current constant value the solution is slowly improved. A reward for parsimony was not used because program bloat was not a problem.

At just sixteen cases per output, the size of the training set used in the experiments was unusually small for EC. However, because the fitness function adopted required a threshold fitness score in all cases, then the candidate function promoted was most likely to be a useful root function which could 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.

Parameter	Value
Representation	linear GP
Maximum string length	5-30 words maximum
Population size	200-1000
Initial population	random
Population	steady state
Selection mechanism	tournament
Crossover rate	0.25
Mutation rate	0.4
No. of generations	20000

Table 1. Parameter values for GP search

Table 1 summarises the various parameter values, or ranges of values, that we employed. Relatively little effort was made to optimise them, however, as the next section demonstrates we were able to obtain usable results which did not seem to be particularly sensitive to changes within the ranges we explored.

4 Washing machine controller problem

This case study followed on from a previous pilot (based on a simple fridge controller) that demonstrated the feasibility of our approach. It targeted a more

complex problem with 14 inputs and seven threads plus it introduced temporal sequencing rather than continuous operation.

– **System inputs**

Timer inputs, TS0 ... TS6 timing sequence inputs, provided by the OS.

Target temperature the desired wash temperature.

Water temperature the current water temperature

Drum Full signals washing drum full of water.

– **System outputs**

Heat drive for the water heater. This output is to be zero if the current water temperature exceeds the chosen temperature.

Hot, Cold Boolean signals to open the respective water fill valves.

Slow, Fast Boolean signals to select drum rotation speed.

Drain Boolean signal to enable the water drain pump.

Table 2 outlines the required functional relationship between the inputs and outputs for the complete operational cycle of the washing machine. Note that not all time periods (TS0 ... TS6) are necessarily equal in duration.

	TS0	TS1	TS2	TS3	TS4	TS5	TS6	DrumFull
Heat (cont.)		$f(\epsilon_T)$						
Heat (logic)		X						
Slow		X	X		X	X		
Fast				X			X	
Cold	X			X				X
Hot	X							X
Drain			X			X	X	
Description	Fill	Wash	Drain	Spin	Rinse	Drain	Spin	

Table 2. Washing Machine Cycle. N.B. $f(\epsilon_T)$ indicates a function of water temperature error.

5 Problem solution results

The main problem encountered in the solution of this problem centred on the Heat output because the evolution of the code for this output required that the desired continuous function relating the temperature error and heat drive be found. In addition, the output had to be clamped to zero in the event that the water was too hot or if the current operational time period was not Wash. This overall output required the evolution of a combined continuous and logical function and this proved to be unachievable when tried. The solution adopted was to make the overall Heat output the composite of two evolved tasks: one continuous and one logical. In this way, the fitness landscape of the boolean

function did not obscure that for the continuous function. In order to keep the whole system general, we suggest that all continuous outputs functions have an associated logical gating function – whether used or not. This gating would be performed by the OS. For example, the combined Heat function here could be formed using the functions:

$$\begin{aligned} \text{Heat (continuous)} &= (\text{Target_temp} - \text{Actual_temp}) * \text{Span_constant} \\ \text{Heat (logic)} &= (\text{Target_temp} > \text{Actual_temp}) \& \text{TSO} \end{aligned}$$

Evolving the washing machine controller problem proved to be relatively straightforward with the evolution of the continuous heat function proving to be the most difficult thread. In order to determine the nature of the relationship between the population size, string length and evolvability, some 900 experiments were performed.

Length	5	10	15	20	25	30	Total
Fail	138	138	144	146	143	142	851
Success	12	12	6	4	7	8	49
Total	150	150	150	150	150	150	900

Table 3. Success rate v. string length

Using longer strings to construct an initial population, one would expect, would generally result in a more diverse population leading to more effective search. However, the results in Table 3 indicate the opposite result. There would seem to be two possible explanations for this unexpected result. First, the evolved program strings were executed sequentially forwards (with the exception that the Branch instruction might cause some instructions to be skipped) and so the contents of the output register would be overwritten by successive output operations. As a consequence, only the code and data directly contributing to the final output operation was relevant, therefore, the overall string length cannot be simply viewed as an indicator of potential. Second, the effect of mutation (and effective search) were diluted in direct proportion to the string length. This is because the probability of selecting the critical instruction for a constructive mutation diminished as the string length increased. These two factors can explain why the counterintuitive observation that string length is negatively proportional to success.

Similarly and, perhaps, even more surprisingly, the experiments also failed to demonstrate any useful advantage in operating with larger population sizes (refer to table 4). Again, the greater diversity afforded by a larger initial population might be expected to 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 [6], but perhaps the dominant factor here is the subsequent loss of diversity.

Pop.	200	400	600	800	1000	Total
Fail	172	173	171	167	168	851
Succeed	8	7	9	13	12	49
Total	180	180	180	180	180	900

Table 4. Success rate v. population size

Due to the fitness assessment measure used the vast majority – or sometimes all – of the candidate solutions in the initial population were 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 [3] in a study where GP and hill climbing (HC) algorithms 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 [11]). 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.

6 Conclusions

In this paper we have identified embedded control programs as a fruitful application domain for GP. We have suggested an approach for using GP to create such programs automatically that differs from other work in that we use the structure inherent in this class of problem to allow a divide and conquer strategy to be effective. The approach has been demonstrated with a relatively complex problem of a washing machine controller involving seven outputs and fourteen inputs. Here the worst thread was soluble about one in every eighteen runs and this took only around half an hour on a three year old PC. The proposed approach proved to be relatively insensitive to variations in the EC parameters used during evolution which we believe to be a positive result for a system aimed at wide application i.e. little or no tuning is required.

There are a number of avenues for future work. Widening the application scope for this system would involve increasing the trustworthiness of such a non-human system. Measures to improve reliability such as schemes whereby an output is the consensus of several (possibly dissimilar) contributors: run time bounds checking of variables (triggering pre-written exception processing within

the general OS) and the use of watchdog timers might all be considered. Similarly, automated testing will enhance confidence in this technique. A ‘front-end’ tool is also needed to capture the problem from the target users so that training and test files may be generated for the EC system. Output could also include parameters values for use during evolution. This could involve graphical, formulaic, algorithmic or truth-table entry of the problem to be tackled.

References

1. Banzhaf, W. Nordin, P. Keller, R. Francone, F. *Genetic Programming: An introduction*. Morgan Kaufmann Publishers Inc. 1998.
2. Friedberg R., “A learning machine”, *IBM J. Research & Development* 1(2) pp2-13, 1958.
3. Fuchs, M. “Large populations are not always the best choice in genetic programming”, *GECCO-99. Proceedings of the Genetic and Evolutionary Computation-Conference*, pp1033-8 vol.2, Morgan Kaufmann Publishers, San Francisco, CA, 1999.
4. Holland J., *Adaptation in natural and artificial systems*. MIT Press, Cambridge MA, 1992.
5. Koza, J. R. *Genetic programming: On the programming of computers by means of natural selection*, Cambridge, MA: MIT Press, 1992.
6. Langdon, W.B. “Convergence of Program Fitness Landscapes”, *GECCO-2003. Proceedings of the Genetic and Evolutionary Computation-Conference*, Chicago, 2003.
7. Langdon W.B. Poli R. *Foundations of Genetic Programming*. Springer-Verlag Berlin and Heidelberg, 2002.
8. Leung, K.S., Lee, K.H. Cheang, S.M. “Parallel programs are more evolvable than sequential programs”, *Genetic Programming, Proceedings of 6th European Conference*. Springer, Berlin. LNCS 2610, pp107-118, 2003.
9. Rechenberg I., *Evolutionstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution*. Frommann - Holzboog Verlag Stuttgart, 1973.
10. 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, pp83-96, 1998.
11. Wolpert, D.H. Macready., W.G. “No Free Lunch Theorems for Search”, *IEEE Transactions on Evolutionary Computation* 1(1), pp67-82, 1997.