Pairwise Sequence Comparison and the Genetic Programming of Iterative Concurrent Programs

Brian J. Ross
Department of Computer Science
Brock University
St. Catharines, Ontario, Canada
bross@cosc.brocku.ca

ABSTRACT

The genetic programming of iterative concurrent programs written in the CCS process algebra is investigated. Using a generational genetic programming scheme, experiments succesfully evolved a cyclic concurrent program that performs a even-parity-2 analysis on a communicating input stream. The fitness evaluation strategy determines how well programs communicate with randomly generated streams of input signals. Fitness is measured by performing a pairwise sequence alignment comparison of two execution sequences – the output sequence generated by the program communicating with the test signal stream, and the correct output sequence for that test case. The optimal edit distance between these sequences is efficiently computed using dynamic programming. The main result is that sequence alignment evaluation against randomly generated test cases is a promising evaluation strategy for evolving cycling protocols.

1 Introduction

This research investigates the applicability of genetic programming (GP) (Koza 1992) towards the evolution of iterative concurrent programs. Iterative programs repeatedly and indefinitely execute so long as a input communication is available to them. They are also concurrent if they execute in unision with a program communicating with them. Concurrency is an important computational paradigm, and is the subject of much theoretical and practical study (Hoare 1985, Milner 1989, Peterson 1977). Although there are no inherent theoretical obstacles in using GP on iterative concurrent programs, there are practical considerations that must be addressed due to the complexities inherent in concurrent computation. A nondeterministic concurrent program can behave differently during separate runs in an identical environment. Multiple concurrent processes produce a combinatorial explosion of possible interactive behaviors when executed together. This contrasts with the sequential and normally deterministic computation studied in most GP systems, in which a given program is associated with one single behaviour when executed in some environment.

This paper shows that it is possible to successfully evolve iterative concurrent programs using a standard GP system. The iterative, nondeterministic nature of the programs, however, means that special strategies for interpreting and evaluating them are required. The combinatorially large execution space of most programs is controlled through execution controls in the concurrent language interpreter, as well as by constraining the execution environment during program evaluation. Fitness evaluation tests each generation of programs on a set of randomly generated sequences of input signals. The testing strategy compares the program's output sequence with the corresponding correct sequence for that test. The fitness metric uses pairwise sequence alignment measurement, as used in genetic DNA analysis, and is efficiently computed using dynamic programming. Although the random test sequences can enforce differing fitness criteria on the corresponding generation's population, the cumulative effect of these criteria across multiple generations produces populations that adhere to the overall required specification.

Section 2 reviews the topics of process algebra and pairwise sequence alignment comparison. An experiment that evolves an iterative concurrent even-parity-2 program is discussed in section 3. Conclusions, future directions and related work are discussed in section 4.

2 Background

2.1 Process Algebra

Process algebra are mathematical systems used for modeling and analyzing concurrent systems (Hennessy 1988, Hoare 1985, Milner 1989). Although they are analytical tools, they have inherent computational semantics, and can be considered to be abstract programming languages. The process algebra used in this paper is a minor variant of CCS (Calculus of Communicating Systems) (Milner 1989). It uses most of the operators of CCS, chosen for their simple syntax and semantics, as well as an iteration operator from the π–calculus (Milner 1991).

$$\mathbf{Iter}\quad \frac{E \xrightarrow{\alpha} E'}{!E \xrightarrow{\alpha} E'|!E} \qquad \mathbf{Sum_j}\quad \frac{E_j \xrightarrow{\alpha} E'_j}{\sum E_i \xrightarrow{\alpha} E'_j}$$

$$\mathbf{Com_1}\quad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \mathbf{Com_2}\quad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$\mathbf{Com_3}\quad \frac{E \xrightarrow{\ell} E' \qquad F \xrightarrow{\overline{\ell}} F'}{E|F \xrightarrow{\tau} E'|F'} \qquad \mathbf{Act}\quad \frac{}{\alpha.E \xrightarrow{\alpha} E}$$

$$\mathbf{Res}\quad \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L} \quad (\alpha, \overline{\alpha} \notin L)$$

Figure 1 Transitional semantics of CCS operators

CCS is an algebra which allows the description and analysis of networks of communicating agents. An *agent* is a mechanism whose behavior is characterised by discrete actions. Agents are described using a set of *agent expressions*, $\mathcal{E}$, recursively constructed as follows (E ranges over $\mathcal{E}$):

Prefix: $\alpha.E$	Choice: $E_1 + ... + E_k$
Composition: $E_1 \mid E_2$	Restriction: $E\backslash L$
Iteration: $!E$	Null: $\mathbf{0}$

The semantics of these operators are defined by the transitional rules of Figure 1. These transitions are sequents in which the expression below the line can be inferred when the expressions above (if any) hold. The expression $E \xrightarrow{\alpha} E'$ represents the transition of agent E into derivative E', generating the action α in the process. When multiple transitions occur, as in $E \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$, then $\alpha_1 \cdots \alpha_n$ an *action sequence* of E. Given an expression E, the transition rules can be repeatedly applied, resulting in a set of possible traces.

Descriptions of the transitions in figure 1 are as follows. (i) **Act**: This describes an agent transition in terms of its immediate actions α. $\mathcal{A}$ is a set of action *names*, and $\overline{\mathcal{A}}$ is the set of *co-names*. The set of *labels* $\mathcal{L}$ is $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$. The set of *actions* Act is $Act = \mathcal{L} \cup \{\tau\}$, where τ is a distinguished silent action. For convenience, $\alpha.\mathbf{0}$ is denoted α. (ii) **Sum**$_j$: In $E_1 + E_2$, E_1 and E_2 are alternate choices of behavior. The notation $\sum E_i$ denotes $E_1 + E_2 + ... + E_k$ for $k \geq 1$. (iii) **Com**$_i$: Agent composition represents how agents behave, both autonomously (**Com**$_1$**,** **Com**$_2$) and interactively (**Com**$_3$). **Com**$_3$ models a *handshake*, which is a simultaneous communication between two agents. In order for a handshake to occur, two agents simultaneously execute identical actions of opposite signs. For example, in $(a.P+\overline{b}.Q) \mid (\overline{a}.R+c.S)$, a communication can occur between $a.P$ and $\overline{a}.R$, and results in a hidden τ. An expression is said to be in *deadlock* if it cannot generate a visible action or τ. Then it is equivalent to the *null* process $\mathbf{0}$, which is inactive. (iv) **Iter**: An iteration operator provides a means for iteration by permitting an indefinite number of agents to be spawned as necessary. (v) **Res**: Restriction hides the specified actions in set L from being observed.

2.2 Sequence Comparison

Sequence comparison algorithms are used to measure the relative similarity between two sequences of symbols (Sankoff and Kruskal 1983, Myers 1991, Giegerich and Wheeler 1996). Such sequences can be from protein or DNA, text files on an operating system, or uttered phrases given to a speech recognition system. The examples in this section are taken from (Giegerich and Wheeler 1996).

The GP evaluation scheme in section 3 uses a pairwise sequence comparison algorithm to compare a program's output sequence and the correct sequence. An edit distance is the total number of edit operations required to change one sequence into the other. The edit operations of interest are: (i) Delete (delete a symbol of a sequence); (ii) Insert (insert a symbol into a sequence); and (iii) Replace (replace a symbol from one sequence into a symbol from the other sequence). Consider the following two sequences:

$$\begin{array}{lcccccccc} s_1: & a & g & c & a & c & a & c & a \\ s_2: & a & c & a & c & a & c & t & a \end{array}$$

There are many ways in which to convert s_1 into s_2. One possible edit sequence comes from the following alignment:

$$\begin{array}{cccccccc} a & g & c & a & c & a & c & - & a \\ a & - & c & a & c & a & c & t & a \end{array}$$

The "$-$" symbol denotes an insertion if it appears in the first (top) sequence, or a deletion if it appears in the second. A replacement is denoted by differing symbols in an alignment. Otherwise, symbols match in the alignment, which requires no edit operation. From this alignment, s_2 is derived from s_1 by deleting g and inserting t, yielding an edit distance of 2. Another possible alignment is the following:

$$\begin{array}{ccccccccc} a & g & - & c & a & c & a & c & a \\ a & c & a & c & a & c & t & - & a \end{array}$$

This alignment involves a replacement (g to c), an insertion (a), a replacement (a to t) and a deletion (c), resulting in an edit distance of 4.

We use a dynamic programming algorithm to find an optimal alignment of two sequences, and hence their minimal edit distance. The minimal edit distance is a good measurement of sequence similarity. The fewer edit operations required to change one into the other, the closer those sequences are from each other. This will be useful during fitness evaluation in section 3, as we can use this measurement to evaluate how closely a program's output behavior compares with a desired output sequence.

A sketch of the dynamic programming algorithm is as follows. Let s_i denote the i^{th} symbol of s. Let *i:s:j* denote the subsequence of s between positions i to j, where position 0 is before the first symbol of a sequence, and position k is after the last symbol of a sequence of size k. Consider two prefixes *0:s:i* and *0:t:j* $(i, j \geq 1)$. Assume we know the optimal alignments of all shorter prefixes of s and t, in particular:

1. *0:s:(i-1)* and *0:t:(j-1)*
2. *0:s:(i-1)* and *0:t:j*
3. *0:s:i* and *0:t:(j-1)*

An optimal alignment of *0:s:i* and *0:t:j* must be an extension of one of the above by: (i) a replacement (s_i, t_j), or match (s_i, t_j), depending on whether $s_i = t_j$; (ii) a deletion (s_i, -); or (iii) an insertion (-, t_j). Dynamic programming chooses the minimum edit distance of the above three cases for each new extension of the sequences compared, summing the cumulative number of edit operations:

$$dist(0:s:i, 0:t:j) = minimum \left\{ \begin{array}{l} dist(0:s:(i-1), 0:t:(j-1)) + \left\{ \begin{array}{l} 1\ if\ \ Replace \\ 0\ if\ \ Match \end{array} \right. \\ dist(0:s:(i-1), 0:t:j+1) \\ dist(0:s:i, 0:t:(j-1)+1) \end{array} \right. \tag{1}$$

This is done by computing a *distance matrix* for the compared sequences, which represents the optimal edit distances for all the sequence prefix combinations. Figure 2 shows an example distance matrix aligning the strings *agcacaca* and *acacacta*. Dynamic programming fills in the table from the top-left to the bottom-right, and eventually the optimal distance of 2 is computed in the bottom-right corner. Therefore, by using previous optimal solutions to obtain new ones, dynamic programming incrementally and efficiently calculates the optimal edit distance. The algorithm is $O(n^2)$.

			s_2							
			a	c	a	c	a	c	t	a
		0	1	2	3	4	5	6	7	8
	a	1	0	1	2	3	4	5	6	7
	g	2	1	1	2	3	4	5	6	7
	c	3	2	1	2	2	3	4	5	6
s_1	a	4	3	2	1	2	2	3	4	5
	c	5	4	3	2	1	2	2	3	4
	a	6	5	4	3	2	1	2	3	3
	c	7	6	5	4	3	2	1	2	3
	a	8	7	6	5	4	3	2	2	2

Figure 2 Example Distance Matrix

3 Experiment: Even-Parity-2

3.1 Problem Description

An even-parity-2 program generates a true signal if an even number of inputs (0 or 2) are true, and otherwise generates false. Unlike a sequential even-parity-2 program which can be supplied its input values as arguments, the concurrent program needs to explicitly receive these values via communication with the environment. Being iterative, the program will need to repeatedly process an indefinite, and possibly nonterminating, number of such input signals.

Consider this CCS expression:

$$(\ \ !Program\ \mid\ t.t.f.t.f.f\cdots\ )\setminus\{t,\ f\}$$

A sequence of t and f signals are to be read by $Program$, and it should generate appropriate output. $Program$ is iteratively executed via the ! operator. The restriction of t and f means that they are not seen as output, but are instead reserved only for communication to $Program$. The protocol to be used is that, for each pair of consecutive t and f actions read, a y and n action (denoting $true$ and $false$) is generated as appropriate. (Note that y and n are visible externally, because they are not restricted). We also specify for the protocol that $Program$ elicits a response as soon as the second action of a pair is read (rather than, for example, read multiple pairs of input, and then generate output for the group). As can be seen in $\mathbf{Com_3}$ in figure 1, the silent action τ results whenever two agents perform a communication. Hence the above expression should generate the following output sequence:

$$\tau\ .\ \tau\ .\ y\ .\ \tau\ .\ \tau\ .\ n\ .\ \tau\ .\ \tau\ .\ y\ \cdots$$

A τ occurs each time $Program$ consumes a t or f from the input stream. The program should cease activity when the stream terminates.

3.2 Genetic Programming System

<u>Parameter</u>	<u>Value</u>
GP scheme:	generational
Tournament selection size:	reprod.=2, replace=3
Functions:	+, .
Terminals:	$t, f, \overline{t}, \overline{f}, y, n$
Fitness function:	pairwise comparison on random input sequences
Population size:	800 (initial=1200)
Maximum generations:	70
Prob. crossover:	0.90
Prob. mutation:	0.05
Prob. internal crossover:	0.90
Prob. terminal mutation:	0.75
Max. depth initial population:	6
Max. depth crossover offspring:	17
Trace limits:	300 unique, 600 overall
Replication limit:	8
Unique population members:	yes

Figure 3 Genetic Programming Parameters

Figure 3 lists various common GP parameters for the experiment. Some of the unique parameters for this system are now described. A generational GP scheme with tournament selection for candidate selection and replacement is used. A generational approach is used so that a new set of test sequences can be generated for each generation. Some parameters such as population size were determined experimentally. Since the initial population can yield many irrelevant programs, culling is performed initially. To simplify search, the minimal set of required CCS operators is used during evolution. Both crossover and mutation are used for reproduction. Crossover is similar to that in (Koza 1992). The mutation operation performs random terminal mutation and larger-scale mutations, both of whose probabilities are parameterized. Large-scale mutations include replacing a program branch by a randomly generated subtree, or other algebraic transformations on CCS expressions. The trace limit parameters control the volume of traces generated during interpretation, in which a maximum of 300 unique traces and 600 unique and duplicate traces are permitted before the interpretation is terminated for an expression. The replication limit is the number of iterations permitted with the "!" operator. The final parameter indicates that programs in the population must be syntactically unique.

Program fitness is evaluated by measuring how well it communicates with a randomly generated set of input sequences. Each sequence is between 4 and 8 actions in length. Variable sized input sequences help discourage the evolution of programs that are hard-wired to read in fixed-length sequences. One such random sequence R might be the following:

$$t\ .\ t\ .\ t\ .\ f\ .\ f\ .\ t$$

A candidate program X is run in concert with the test sequence $R_k (1 \leq k \leq 10)$:

$$(\ X\ \mid\ R_k\ ) \setminus \{t,\ f\} \tag{2}$$

and the set of generated output traces T_k are recorded. For the above expression, the solution trace S_k is:

$$\tau\ .\ \tau\ .\ y\ .\ \tau\ .\ \tau\ .\ n\ .\ \tau\ .\ \tau\ .\ n$$

A correct program's output should correspond exactly to this solution trace, which is called *strong bisimilarity* in (Milner 1989). The minimal edit distance of each resulting trace with solution S_k is determined, and the overall average for that test case R_k is computed:

$$Score_k = \frac{\sum_{i=1}^{|T|} mindist(T_i, S_k)}{|T|} \tag{3}$$

The overall score for a program is the sum of all 10 test case scores:

$$Score = \sum_{i=1}^{10} Score_i \tag{4}$$

Should the interpreter pre-emptively cancel an interpretation due to trace overflow, the result will be an empty trace. Empty traces are not given to the dynamic programming procedure, but are instead given the score 100, in order to penalize expensive expressions.

A hypothesis being tested in this paper is that heterogeneous fitness criteria throughout multiple generations can still produce homogeneous characteristics in the population in the longterm. At the start of each new generation, a set of 10 random test cases, along with their solution traces, is created. They are used for the rest of that generation's processing, and are replaced at the beginning of the subsequent generation. This means that the fitness criteria for each generation is unique: a fit program as evaluated by the tests in one generation may prove much less fit in the next generation.

The use of restriction in equation 2 above is beneficial, as it reduces the number of traces generated during interpretation. Because t and f are restricted, a subexpression in a program will cease contributing to traces when a $\overline{t}$ or $\overline{f}$ action arises without handshaking with a corresponding t or f signal in the input. Terms containing such extraneous communications can be considered to be a form of *intron*. In addition, introns arise in the form of duplicated terms arising during evolution.

3.3 Experimental Results

Figure 4 Fitness curves (avg 25 runs)

Figure 4 plots the average, best, and best simplified (described below) fitness curves, averaged for 25 runs. All but 3 of the 25 runs found solutions within 70 generations. Some example programs from one successful run are as follows. (Note that the fitness values are relative to the particular random test cases used in that particular generation.) In the initial population, this best program had a fitness of 26.75:

$$(\overline{f} + \overline{t}) + \overline{f}.(\overline{f} + y)$$

At generation 32, the following best program had a fitness of 7.3333:

$$\overline{f}.(\overline{t}.n + \overline{f}.y + \overline{t}.y) + \overline{t}.((\overline{f}.y + \overline{t}.y) + (\overline{f}.y + \overline{f}.n) + \overline{f})$$

Finally, at generation 63, a solution was obtained:

$$\overline{f}.(\overline{f}.y + \overline{t}.n) + \overline{t}.(\overline{f}.n + \overline{t}.y)$$

This is an optimal CCS solution for the problem.

Many solutions obtained during the 25 runs corresponded to the minimally-sized correct solution. Others, however, were incorrect, even though their fitness scores were 0.0. For example, the program

$$\overline{f}.(\overline{t}.n+\overline{f}.((\overline{t}.(\overline{t}+\overline{f}.\overline{f})+y)+y+\overline{t}))+\overline{t}.(\overline{f}.n+\overline{t}.y) \qquad (\dagger)$$

has a trace set

$$\{\overline{f}.\overline{t}.n,\ \overline{f}.\overline{f}.y,\ \overline{t}.\overline{f}.n,\ \overline{t}.\overline{t}.y,\ \overline{f}.\overline{f}.\overline{t},\ \overline{f}.\overline{f}.\overline{t}.\overline{t},\ \overline{f}.\overline{f}.\overline{t}.\overline{f}.\overline{f}\}.$$

in which the first 4 terms are correct, but the last 3 are not. The GP algorithm stopped, however, given its fitness score of 0.0. When evaluated with a different set of traces, it yielded a score of 2.0, due to the extraneous traces in the above set. This occurred because a generation's set of 10 test cases for a run cannot be guaranteed of being complete. Completeness in the context of our experiment is more than covering all known $\{t, f\}$ combinations in a truth table. Rather, a complete set of tests must exercise all the internal subexpressions inherent in a given CCS program. Typically, programs identified as "false solutions" included within their trace sets the correct even-parity-2 case processing behavior, but also performed extraneous behaviors. Such programs tricked a particular suite of tests by handling all the tests correctly, because the erroneous behavior is not exercised by the tests, and is furthermore hidden by restriction when handshaking does not elicit their activity. For example, in the expression

$$(\ (\overline{f}.\overline{t}.n+\overline{t}.\overline{f}.\overline{t})\ \mid\ f.t\ )\setminus\{t,\ f\}$$

only the "$\overline{f}.\overline{t}.n$" term is exercised by the test "$f.t$". The "$\overline{t}.\overline{f}.\overline{t}$" term is restricted, because its prefix $\overline{t}$ is restricted.

The incompleteness of random test cases can be overcome in a number of ways. Increasing the size of input traces and number of test cases will raise the probability of a complete test suite for a program, and result in more inter-generational stability with respect to fitness evaluation. This will adversely impact computation time. Another approach is program transformation. Since there are a finite number of terms in a CCS program, one can simply find the terms that are extraneous, and remove them. This can be done using the following procedure:

Loop until all choice terms have been tested:
– delete a term t from program
– reevaluate the simplified program's fitness
– if the program is more fit than previously
then leave term t out

This procedure should be done with a number of separate sets of test streams, due to the fact that different streams exercise different aspects of the program's behavior. If the program still has a perfect fitness with respect to a number of new test sequences, it is highly likely to be correct. Correctness cannot be ensured, however, unless a rigorous analysis of the program is undertaken, for example, examining its trace set directly. Our experience was that all false solutions were successfully converted into true solutions using the above simplifying transformation, and with the use of merely one new set of test streams. For example, the above false solution (†) is simplified to:

$$\overline{f}:(\overline{t}:n+\overline{f}:(y+y))+\overline{t}:(\overline{f}:n+\overline{t}:y)$$

which is verifiably correct. Moreover, many solutions can evolve much earlier if this simplification scheme were to be regularly used, but at the expense of processing time (see the "Best simplified" curve of Figure 4). It may be unwise to do such simplifications too liberally, because extraneous terms do contribute to genetic diversity during formative phases of evolution.

3.4 Implementation Details

The GP system is implemented in Quintus Prolog 3.2 on the Silicon Graphics Irix 6.2 environment. The core of the CCS interpreter takes the form of 11 Prolog clauses that are nearly literal translations of the transition rules in Figure 1. The implementation of the iteration ("!") operator, however, is refined with respect to its transition rule. That rule specifies that a spawned agent ($E \mid !E$) can perform internal interleaving via "|". This interleaving is irrelevant and prohibitively expensive; we only require a spawned agent to communicate with actions in the input stream, and not with copies of itself. We therefore prohibit internal interleaving within ! terms.

Complete runs of the GP system take between 3 to 5 hours, and most of this time is dedicated to fitness evaluation. The Prolog implementation is admirably fast considering the considerable processing that occurs during evaluation. Each CCS program is run against 10 test cases. With the parameters in Figure 3, each test case may generate upwards of 300 unique traces, beyond which its interpretation is cancelled. Hence a worst case is 3000 trace comparisons for a program, each of which is pairwise compared to the correct output sequence using dynamic programming. Realistically, most pre-solution programs generate perhaps 20 traces per test case; the solution program should generate one.

4 Conclusion

This research extends previous work that applies GP towards non-iterative CCS programs (Ross 1998). There, the raw trace sets from CCS programs were scored according to a number of qualitative criteria, such as trace set size, trace size, action orderings, missing actions, and others. Engineering effective fitness functions from different criteria is often challenging and *ad hoc*. This paper shows that comparing programs to randomly generated sequences of input is a promising alternative. Rather than creating a clever set of complete and correct tests, one allows nature to influence the direction of evolution. The use of pairwise sequence comparison metrics proved to be a practical heuristic in this regard. We exploited the fact that heterogeneous fitness criteria can be used across multiple generations to evolve homogeneous population characteristics. This is not new, as many genetic algorithms use dynamically adaptive fitness functions. As discussed in section 3.3, one issue when using randomly sampled test cases is that they are not guaranteed of being complete with respect to a given program. In the case of our experiment, a simplifying transformation scheme overcame this problem. More research is needed to see how well this strategy scales upwards to more complex concurrent protocols. It is worth investigating possible relationships to PAC learning and evolution (Valiant 1984), as it might indicate some metrics regarding the size of streams and test cases necessary to obtain solutions within a desired level of efficiency and accuracy.

The iteration operator permits unrestricted iteration, which is problematic for most GP systems. We solved this problem by limiting the iteration to 8 levels, and this had no ill effect on the generality of the iterative solutions. This is similar in spirit to other research in recursive GP in which unrestricted recursion is controlled via finite data structure consumption during recursive calls, or recursion counting and pre-emption (Brave 1996, Nordin and Banzhaf 1995, Wong and Leung 1996).

Process algebra are a form of finite automata, and so our goals are related to GP and GA work in grammatical language induction (Dunay *et al.* 1994). The subset of the CCS language used in this paper is less expressive than a regular grammar; future work will extend the algebra to permit iterative subexpressions within an expression. Other work has used GP towards evolving network-like structures that map to parallel systems (Poli 1997), and the autoparallelisation of sequential programs (Walsh and Ryan 1996). GA's have been applied to distributed and parallel systems (Allan and O'Neill 1994, Harp and Samad 1996, Seredynski and Frejlak 1993).

References

Allan, V.H. and M.R. O'Neill (1994). Software Pipelining: A Genetic Algorithm Approach. In: *Parallel Architectures and Compilation Techniques* (M. Cosnard, G.R. Gao and G.M. Silberman, Eds.). Elsevier. pp. 311–314.

Brave, S. (1996). Evolving Recursive Programs for Tree Search. In: *Advances in Genetic Programming II* (P.J. Angeline and K.E. Kinnear, Eds.). pp. 203–219. MIT Press.

Dunay, B.D., F.E. Petry and B.P. Buckles (1994). Regular Language Induction with Genetic Programming. In: *Proc. 1st IEEE Conference on Evolutionary Computation*. pp. 396–400.

Giegerich, R. and D. Wheeler (1996). Pairwise Sequence Alignment. Technical report. VCNS Biocomputing Division, U of Bielefeld (Germany).

Harp, S.A. and T. Samad (1996). Genetic Synthesis of Neural Network Architecture. In: *Handbook of Genetic Algorithms* (L. Davis, Ed.). Thomson Computer Press.

Hennessy, M. (1988). *Algebraic Theory of Processes*. MIT Press.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice–Hall.

Koza, J.R. (1992). *Genetic Programming*. MIT Press.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Milner, R. (1991). The Polyadic π-Calculus: A Tutorial. Technical Report ECS-LFCS-91-180. LFCS, U. of Edinburgh.

Myers, E.W. (1991). An Overview of Sequence Comparison Algorithms in Molecular Biology. Technical Report TR 91-29. Dept of Computer Science, University of Arizona.

Nordin, P. and W. Banzhaf (1995). Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code. In: *Proc. 6th Intl. Conf. on Genetic Algorithms*. Morgan Kaufmann.

Peterson, J.L. (1977). Petri Nets. *Computing Surveys*.

Poli, R. (1997). Evolution of graph-like programs with parallel distributed genetic programming. In: *Proc. 7th Intl. Conf. on Genetic Algorithms* (Thomas Back, Ed.). Morgan Kaufmann. pp. 346–353.

Ross, B.J. (1998). The Evolution of Concurrent Programs. *Applied Intelligence (journal)*. In press.

Sankoff, D. and J.B. Kruskal (1983). *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley.

Seredynski, F. and P. Frejlak (1993). Genetic Algorithms Implementation of Process Migration Strategies. In: *Intl. Conf. on Parallel Computing* (G.R. Joubert *et al*, Ed.). North-Holland. Grenoble, France. pp. 337–344.

Valiant, L.G. (1984). A Theory of the Learnable. *Communications of the ACM* **27**(11), 1134–1142.

Walsh, P. and C. Ryan (1996). Paragen - A novel technique for the autoparallelisation of sequential programs using GP. In: *Proc. Genetic Programming 1996* (J.R. Koza *et al*, Ed.). MIT Press. pp. 406–409.

Wong, M.L. and K.S. Leung (1996). Learning Recursive Functions from Noisy Examples using Generic Genetic Programming. In: *Proc. Genetic Programming 1996*. MIT Press. pp. 238–246.