Applied Intelligence, 8, 21–32 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

The Evolution of Concurrent Programs

BRIAN J. ROSS

bross@cosc.brocku.ca

Department of Computer Science, Brock University, St. Catharines, ON, Canada L2S 3A1

Abstract. Process algebra are formal languages used for the rigorous specification and analysis of concurrent systems. By using a process algebra as the target language of a genetic programming system, the derivation of concurrent programs satisfying given problem specifications is possible. A genetic programming system based on Koza's model has been implemented. The target language used is Milner's CCS process algebra, and is chosen for its conciseness and simplicity. The genetic programming environment needs a few adaptations to the computational characteristics of concurrent programs. In particular, means for efficiently controlling the exponentially large computation spaces that are common with process algebra must be addressed. Experimental runs of the system successfully evolved a number of non–iterative CCS systems, hence proving the potential of evolutionary approaches to concurrent system development.

Keywords: genetic programming, process algebra, CCS, concurrency

1. Introduction

Genetic programming (GP) is an evolutionary learning technique used for the automatic synthesis of computer programs [5]. Using the metaphor of Darwinian "survival of the fittest", a population of programs constructed in a programming language of interest are bred and replicated according to some fitness criteria. Programs that are fitter will survive and breed, while weaker programs will perish. After a number of generations, a program that satisfies the fitness criteria of the problem will hopefully arise.

Genetic programming has been successfully used to derive sequential programs. Lisp is typically used in GP work, due to its symbolic basis, and its efficiency on single processor computers. Although Koza states that genetic programming will work for any programming language of interest, relatively little work has applied it towards programming languages outside of Lisp. Other programming paradigms which have been used include RISC machine language [8], object–orientation [2], C [14], and Prolog [14].

Concurrent and parallel programming continues to be an active research area. Multiprocessor systems are now common in multi–user servers, high–performance workstations, and supercomputers. Concurrent computations are inherently more complex than conventional sequential ones, and writing correct and efficient concurrent programs can be difficult to do. As a consequence, much research effort has focussed on the formal derivation and analysis of concurrent computations, for example [10, 9, 1, 3, 7]. To this end, process algebra have been derived. Process algebra are mathematical calculi used for modelling concurrency. Their strengths include methods for abstraction, rigorous language definitions, and robust theories of equivalence.

This paper uses a process algebra as a target language for GP. The GP system requires some adaptation to handle the concurrent paradigm as modelled by process algebra. Concurrent programs can be highly

nondeterministic, and process algebra model this nondeterminism exhaustively. The GP system must deal with a large syntactic program space (as with Lisp), as well as combinatorially large computation spaces for single concurrent programs (unlike Lisp). One challenge is to accommodate the often intractable nature of nondeterministic computations, without overly inhibiting acceptable nondeterministic characteristics in the population.

An overview of the CCS process algebra is given in section 2. A few specific issues arising with the evolution of concurrent systems are discussed in section 3. Section 4 discusses the genetic programming system. Some examples runs that evolve concurrent parity functions and scheduler programs are presented in section 5. A discussion concludes the paper in section 6.

2. Process Algebra and CCS

The process algebra used is Milner's Calculus of Communicating Systems, or CCS [7]. It is selected because of its minimal set of primitives, its rigorous transitional semantics, and its popular acceptance in the research community. Since the problems studied in this research are non-iterative, we consider CCS without recursion. Please see [7] for an in-depth discussion.

CCS is an algebra which allows the description and analysis of networks of communicating agents. An *agent* or *process* 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 $\mathcal{E}$):

$\alpha.E$	Prefix
$\sum_{i\in I} E_i$	Summation
$E_1 \mid E_2$	Composition (or Interleaving)
$E\backslash L$	Restriction
$E[f]$	Relabelling
$\mathbf{0}$	Null

Milner defines the semantics of the other equational operators using the transitional rules of Table 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 Table 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) $\mathbf{Sum}_j$: 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) $\mathbf{Com}_i$: Agent composition represents how agents behave, both autonomously ($\mathbf{Com}_1$, $\mathbf{Com}_2$) and interactively ($\mathbf{Com}_3$). $\mathbf{Com}_3$ 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) **Res**: Restriction hides the specified actions in set L from being observed. (v) **Rel**: A *relabelling function* f renames actions.

A significant part of CCS theory is devoted to various concepts of behavioral equivalence. A *bisimilarity* is an observed equivalence amongst agents. One bisimilarity practical for this paper is *observation equivalence*, which is denoted by the $\approx$ equivalence relation. Let $A \stackrel{\hat{\alpha}}{\Rightarrow} A'$ represent the transition of A into A' where the stream $\hat{\alpha}$ has all "τ" actions removed. Then $P \approx Q$ iff, for all $\alpha \in Act$,

(i) Whenever $P \xrightarrow{\alpha} P'$, then for some Q',
$Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$, and $P' \approx Q'$.
(ii) Whenever $Q \xrightarrow{\alpha} Q'$, then for some P',
$P \stackrel{\hat{\alpha}}{\Rightarrow} P'$, and $P' \approx Q'$.

This states that agents with identical initial transitions and derivatives are considered congruent. Another bisimilarity is *trace equivalence*. Two agents are trace equivalent if they generate the same visible actions after all τ's are removed. Further discussion of equivalence is in the next section.

Table 1. Transitional semantics of CCS operators

Act $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$	$\mathbf{Sum_j}$ $\dfrac{E_j \xrightarrow{\alpha} E'_j}{\sum E_i \xrightarrow{\alpha} E'_j}$	**Res** $\dfrac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L} \ (\alpha, \overline{\alpha} \notin L)$
$\mathbf{Com_1}$ $\dfrac{E \xrightarrow{\alpha} E'}{E \vert F \xrightarrow{\alpha} E' \vert F}$	$\mathbf{Com_2}$ $\dfrac{F \xrightarrow{\alpha} F'}{E \vert F \xrightarrow{\alpha} E \vert F'}$	$\mathbf{Com_3}$ $\dfrac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\overline{\ell}} F'}{E \vert F \xrightarrow{\tau} E' \vert F'}$
	Rel $\dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$	

3. Some concurrency issues

3.1. Observing concurrent program behavior

When a finitary CCS program is executed, the process may generate an indefinite sized stream, a number of different streams, or no activity at all. CCS does not use a notion of time, and hence expression efficiency and the relative time interval between actions is not modeled. This implies that an observer cannot ascertain if the expression has terminated, unless a termination convention is established [4], or the user has access to the internal state of the CCS interpreter (which is unreasonable). A means for identifying the end of a trace is therefore required.

CCS is a nondeterministic language, and nondeterminism arises from the "+" and "|" operators. Nondeterminism means that separate executions of a given CCS program may result in differing output. Many problems require the evaluation of all possible behaviors of a CCS program, while others may only require that a program behaves well in most circumstances. The transitional semantics of CCS models all possible execution behaviors of expressions. If a CCS expression is being interpreted, then the interpreter can execute the expression exhaustively, and return a set of traces which the expression will generate. Of course, successfully accomplishing this depends on the size of the finite trace space, which may be exponentially large with respect to the expression size, and hence impractical to obtain and process.

Of practical significance to GP is the choice of a suitable equivalence relation to use within evaluation functions. One convenient way of analysing CCS programs is by checking for a bisimilarity between the candidate program and some predefined test cases. These tests may elicit both positive and negative behaviors for the fitness score. Test cases may be implemented as CCS expressions that interact with the program being evaluated, or explicit trace sets that are compared to the trace output of the program.

The evolution of CCS programs is more efficient if the equivalence used is as unrestrictive as possible. Trace equivalence and observation equivalence are two such bisimilarities. With trace equivalence, intermediate τ's can be discarded from traces. This greatly prunes the trace space, permitting more efficient evaluation of expressions, as well as widening the window of possible solutions within the search space. The use of coarser bisimilarities, however, means that behavioral precision is lost during evaluation, and the programs evolved might not have the nondeterministic properties necessary in more discriminating applications. The final decision of this issue entirely depends upon the problem application.

3.2. Evaluation Function Strategies

When designing a fitness function for CCS program evolution, special attention must be given to the nonterminals selected for the problem. For example, a nondeterministic program may be needed in which interleaving via "|" is the most natural mechanism for nondeterminism. If the evaluation function unduly penalizes programs having a large number of traces, perhaps by tallying into the score the cardinality of the trace set, then programs with "|" may be harshly penalized, and will perish from the population. To avoid this problem, care must be taken that evaluation scores do not discriminate against the nonterminals being used, while balancing with the need for identifying and avoiding intractable programs.

Programs being analyzed may be embedded within a *program wrapper* or output interface [5], which is a

program expression acting as a context or environment within which to test the behavior of a sub–program. Typically, a single expression will be executed consecutively within a number of wrappers, in order to precisely test it in a variety of contexts. Most importantly, program wrappers give the GP system a specialized context within which to evolve the population, and hence focus the search.

When constructing wrappers within evaluation functions, case must be taken that the wrapper does not too generously underspecify the problem. For example, consider this wrapper that tests expression E:

$$(E \quad | \quad (\overline{a}.x + \overline{b}.y)) \setminus \{a, b, x, y\}$$

This wrapper is intended to test for the following behavior: *When* E *receives* $\overline{a}$*, it generates* $\overline{x}$*, or else when* E *receives* $\overline{b}$*, it produces* $\overline{y}$. The terminals in E are $\{a,\ b,\ \overline{x},\ \overline{y}\}$. Note that this terminal set prohibits internal communication in E: no τ's can be generated within E itself, since a communication requires the synchronization of both an action and its co-action. Restriction hides all extraneous traces not conforming to the above desired results, since without restriction all the non–communicating traces between E and the rest of the expression will be produced. If this wrapper is interpreted, the desired output will be seen via the trace "$\tau.\tau$" – both terms on the right-hand side will generate this trace. Care must be taken that the other two possible traces, "τ" and null, are checked for as well. The first represents an instance in which E accepts one or both of a or b, but then does not respectively generate $\overline{x}$ and/or $\overline{y}$. A null trace means E has deadlocked, and does not communicate with the rest of the expression at all.

The problem in the above approach, however, is that the expression E may be the following:

$$a \ | \ b \ | \ \overline{x} \ | \ \overline{y}$$

This expression generates all permutations of the terminal set, and when placed within the above wrapper, perfectly satisfies the intended test. Deadlock does not arise. Unfortunately, this expression is a degenerate solution that satisfies *all* similarly constructed test environments. The problem arises because this test environment is underspecifying the problem: restriction is hiding too many behaviors within E, and hence the problem is trivially satisfied. Note that this generic solution is syntactically simple, and is likely to evolve with genetic programming.

A preferable strategy is to use wrappers like the above in concert with additional analyses of the trace output of the program. A simple heuristic that penalizes overly–large trace sets of E will effectively prohibit exhaustive interleaving resulting from "|". Incorporating such a penalty permits the wrapper to straight–forwardly test E for the desired communicative behavior, while the trace penalty prevents trivial solutions.

4. A Genetic Programming Environment for CCS

The genetic programming algorithm used is in Table 2, and is modeled on Koza's system [5]. The initial population at step 1 uses a ramped half–and–half scheme for random expression generation. Fitness–proportional selection is done. Oversampling is permitted in which the initial population is made larger than that used in subsequent generations. A minor design decision taken to reduce premature convergence is in step 2b(iii). When a child created via crossover exceeds the syntactic depth limit, that child is disregarded, and no contribution is made to the population by the parent or child. This differs from Koza's approach, in which the parent is replicated into the population.

CCS's simple syntax is well–suited for manipulation during reproduction. Figure 1 shows a CCS expression's parse tree, in a similar format to a Lisp S-expression. During crossover, grammatical correctness is preserved by selecting crossover nodes that will result in syntactically valid offspring (Figure 2).

CCS's transitional semantics in Table 1 are a means for directly implementing a CCS interpreter for use by the fitness function. The transitional rules concisely model the behavior of the operators, and all the rules taken together form the basis of an implementable semantics for the language.

A variety of standard control parameters are used, and will be seen in section 5. A few parameters address the computational characteristics of the CCS language. Concurrent computations can generate a combinatorial explosion of different output behaviors. In addition, a vast number of interpretation paths can generate a single trace. A trace limit parameter permits the user to control the interpretation of CCS expressions as used by evaluation functions. Two values can be controlled – the maximum number of unique traces, and the maximum number of all traces (both unique *and* duplicated).

Table 2. Genetic Programming Algorithm

1. Generate initial population
2. **Loop while** current generation $<$ Max generation **and** normalized fitness of best so far $>$ error tolerance

   Loop while next generation population size $<$ Max population size

   Select a genetic operation:

   (a) $P_r \longrightarrow$ Replication:
      i. Select one individual based on fitness.
      ii. Perform replication.
      iii. Add new individual to new population.

   (b) $P_c \longrightarrow$ Crossover:
      i. Select two individuals based on fitness.
      ii. Perform crossover.
      iii. **If** offspring depth $\leq$ maximum depth **Then** add offspring to new population.
3. **If** solution found **Then** output "success".
   Print best solution obtained.

Table 3. Parameters: Parity-2

Parameter	Value
Functions	$+, \vert, .$
Terminals	$x, \overline{x}, t, f$
Initial population size	750
Population size	500
Maximum generations	50
Probability replication	0.10
Probability crossover	0.90
Probability internal crossover	0.90
Max. depth initial population	6
Max. depth crossover offspring	17
Error tolerance	0.001
Trace limit	50 unique, 100 total
Simplify offspring	off

Should a trace limit be reached, the interpreter will return an empty set in response, which lets the evaluation function know that an expression was pre-empted. One other parameter permits a suite of simplifying transformations to be carried out on expressions.

5. Examples

The problems studied in this section typically use the prefix, composition, and choice operators as nonterminals. Other CCS operators are often used in the evaluation functions. The full set of CCS operators could be conceivably used as nonterminals if needed, but were not used in the interest of parsimony.

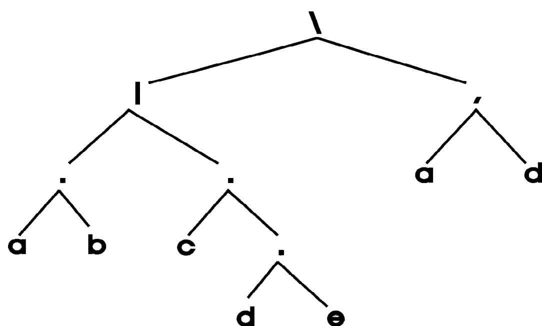

Fig. 1. Grammar tree for: $(a.b \mid c.d.e) \setminus \{a,\ d\}$.

5.1. Parity

The even–parity function is a problem studied by Koza [5]. It is necessary to reimplement the problems for CCS since, unlike Lisp, CCS does not have logical expression primitives. Rather than evaluate Boolean expressions, the CCS expression will instead read in a sequence of input values denoting true and false input signals. After reading the required number of inputs, the appropriate parity output will be generated.

5.1.1. Even-Parity-2 The even–parity–2 function (or equivalence function) generates *true* if both inputs are *true* or *false*, or else outputs *false*. The parameters for this problem are in Table 3. The terminals x and $\overline{x}$ represent the input's *true* and *false* values respectively. The input will be read sequentially as two consecu-

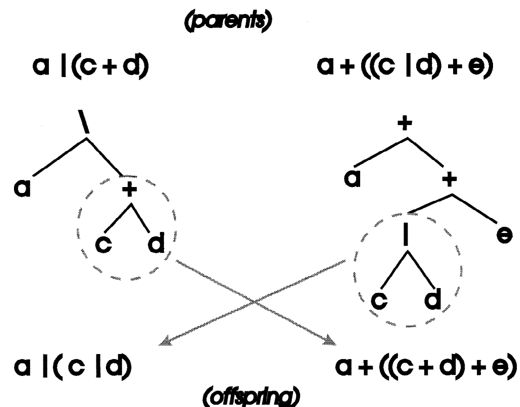

Fig. 2. Crossover of two CCS expressions.

tive signals. For example, $x.x$ is the transmission of two *true* signals. The terminals t and f denote the functions output of *true* and *false* respectively. The intended solution program E should run in the following environment:

$$(E \ \mid \ P) \setminus \{x\}$$

where process P generates a sequence of size 2 of x and $\overline{x}$, and E returns t or f appropriately. For example,

$$(E \ \mid \ \overline{x}.\overline{x}) \setminus \{x\} \xrightarrow{t} \mathbf{0}$$

One successful evaluation strategy is as follows. The entire set of traces from an individual E is obtained from the interpreter. Since trace equivalence will be used, all the τ's are filtered from the traces. This is necessary, because the x and $\overline{x}$ actions may interact via "|" within E. Duplicate traces are also removed. These filtered traces are then compared to the truth table of desired function behavior:

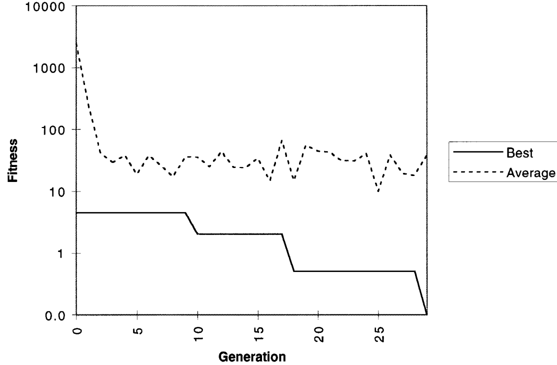

Fig. 3. Fitness curve, Even–parity–2.

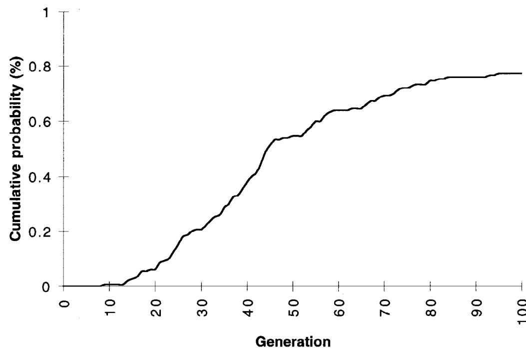

Fig. 4. Cumulative probability of success, Even–parity–2.

I_1	I_2	Out
$\overline{x}$	$\overline{x}$	t
$\overline{x}$	x	f
x	$\overline{x}$	f
x	x	t

The overall score is computed as:

$$Score = \begin{cases} 100 & (if \ \ trace \ \ limits \ \ exceeded) \\ \frac{1}{2}(4 - \#hits + \#misses)^2 \end{cases}$$

Expressions that exceed the trace limit are penalized, which effectively removes them from the population. Otherwise, the number of hits (maximum of 4) are counted and subtracted from the sum, and the number of misses (arbitrary high) are added to it. The value 4 ensures the sum does not go negative, and also penalizes trace sets that have fewer than 4 misses in total. Finally, squaring this sum creates a polynomial that is weighted towards better individuals, and scaling by a half makes the scores more manageable.

The fitness curve for a run of the above problem is in Figure 3. In this and other fitness graphs in this paper, the vertical axis uses a logarithmic scale in order to accommodate both the best and average scores. One of the best individuals of the initial generation was the program

$$x.x.t$$

which had a fitness score of 4.5. On the other hand, a bad program with a fitness score of 5000.0 happened to be

$$(\overline{x}.\overline{x} \ \mid \ \overline{x} \ \mid \ 0) \ \mid \ (0 \ \mid \ \overline{x}) \ \mid \ (0 + f)$$

The best individuals in the next 8 generations had a similar score of 4.5. In generation 10, the fittest program

$$\begin{aligned} &\overline{x}.x.f + x.x.((t + \mathbf{0}) + \mathbf{0}) + \mathbf{0} \\ &\approx \ \ \overline{x}.x.f + x.x.t \end{aligned}$$

evolved, with a fitness of 2.0 (the right–hand term is a simplified version of the actual evolved program on the left). No stronger individual was seen until generation 18, in which

$$\begin{aligned} &\overline{x}.x.f + x.x.(t + \mathbf{0}) + \overline{x}.\overline{x}.(t + \mathbf{0}) \\ &\approx \ \ \overline{x}.x.f + x.x.t + \overline{x}.\overline{x}.t \end{aligned}$$

appeared with a fitness of 0.5. This score continued to be the best until generation 29 found the solution

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

As can be seen, a tier of progressively fitter programs evolved, in which an additional number of correctly handled terms were included in successively fitter generations. Typically, populations would have multiple superlative individuals that covered different terms of the truth table. Crossover merged these terms into offspring that combine these different terms, until the solution handling all four finally arose.

Interestingly, the "|" operator tended to quickly evolve out of the population. This is because the scoring mechanism penalizes large trace sets, which is a likely phenomenon when this operator is present. In this problem, however, the presence of interleaving did not burden the search. A few runs were done that excluded interleaving as a nonterminal, and solutions were obtained in generations 38 and 47.

Figure 4 shows a graph of the cumulative probability of finding a solution to the even–parity–2 problem. The probability of success asymptotically reaches approximately 80% after generation 100, which means there is a 20% chance that a solution is unlikely for any run.

5.1.2. Even-Parity-3 The even–parity–3 problem is considerably more complex. The parameters used are in Table 4. Besides increasing the population size and maximum generation limit, expression simplification was turned on. Early runs with unsimplified expressions resulted in a population full of large expressions containing many irrelevant terms, and in particular, deeply nested variations of "$+\mathbf{0}$" and "$\mathbf{0}+$". Simplification removed the null process from expressions.

The evaluation function was similar to the even–parity–2 case. The following truth table was used:

I_1	I_2	I_3	Out
$\overline{x}$	$\overline{x}$	$\overline{x}$	t
$\overline{x}$	$\overline{x}$	x	f
$\overline{x}$	x	$\overline{x}$	f
$\overline{x}$	x	x	t
x	$\overline{x}$	$\overline{x}$	f
x	$\overline{x}$	x	t
x	x	$\overline{x}$	t
x	x	x	f

Table 4. Parameters: Parity-3

Parameter	Value
Functions	$+, .$
Terminals	$x, \overline{x}, t, f$
Initial population size	1200
Population size	750
Maximum generations	200
Probability replication	0.10
Probability crossover	0.90
Probability internal crossover	0.90
Max. depth initial population	6
Max. depth crossover offspring	17
Error tolerance	0.001
Trace limit	off
Simplify offspring	on

Fitness is computed similarly:

$$Score = \begin{cases} 100 & (if\ trace\ limits\ exceeded) \\ \frac{1}{2}(8 - \#hits + \sum(|miss|))^2 \end{cases}$$

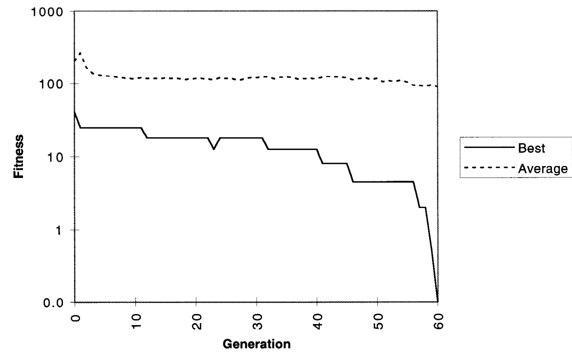

Fig. 5. Fitness curve, Even–parity–3.

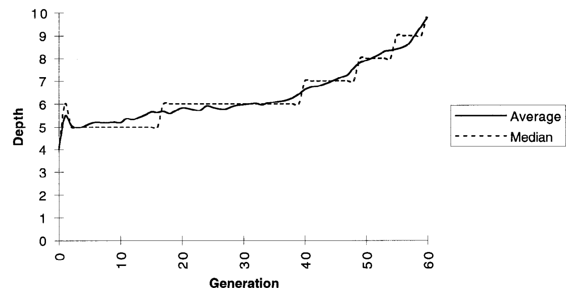

Fig. 6. Structural complexity, Even–parity–3.

The difference is that, instead of tallying the number of misses, the size of each trace that is a miss is summed. This penalizes long terms, resulting in populations with simpler expressions.

The fitness curve of one run that evolved a solution is in Figure 5. The solution obtained in generation 60 is:

$$(\overline{x}.x.\overline{x}.f + (x.\overline{x}.x.t.0 + \overline{x}.\overline{x}.x.f + x.\overline{x}.\overline{x}.f.0)$$
$$+x.(\overline{x}.x.t + x.x.f.0) + (\overline{x}.\overline{x}.\overline{x}.t + x.x.x.f)$$
$$+x.x.\overline{x}.t.0) + x.\overline{x}.x.t + \overline{x}.x.x.t.0 + x.\overline{x}.x.t.0$$

Note that there are 12 terms here, when only 8 suffice, as the evaluation function did not penalize duplicates. This solution E could then be used as follows:

$$(E \quad | \quad x.x.\overline{x}) \setminus \{x\} \xrightarrow{t} \mathbf{0}$$

Figure 6 shows how the structural complexity of the population, measured in expression tree depth, increases as evolution commences.

5.1.3. Scalability of Parity-K In order to measure how well the parity–K experiments scale upwards for higher values of K, the best fitness for 35 runs was recorded and averaged, for each K value of 2 through 5 (see Figure 7). The maximum number of generations is 75. The performance of the K=4 and K=5 runs is not good. This reflects Koza's experience with higher–order parity experiments. In fact, Koza did not obtain a solution for the parity–5 case. Given that the parity–K experiments performed using CCS are more difficult than the Lisp equivalents, the results of Figure 7 are not surprising.

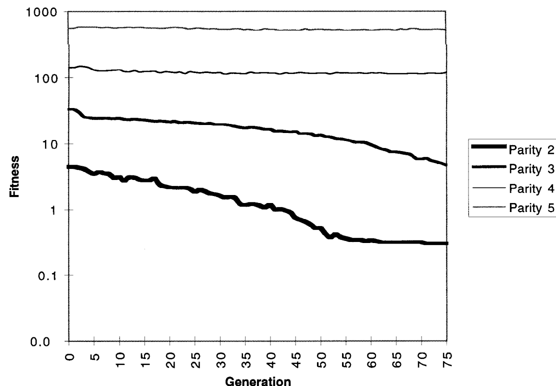

Fig. 7. Average best fitness for Parity–K (35 runs).

Higher–order parity–K problems are feasible for genetic programming if alternate evolution strategies are considered. For example, Koza used automatic function definition to obtain more satisfactory performance. Alternately, Figure 8 plots the average best fitness for 35 runs of parity–K problems, using a steady–state GP system with tournament selection. The performance is considerably better than in the previous graph. The parity–2 and parity–3 runs all found solutions by generations 21 and 37 respectively. There is a noticeable improvement in the K=4 runs, and even the K=5 runs have a downward trend. Further improvements are likely if ADFs were to be used as well. The conclusion from these scalability experiments is that the CCS language is not a contributing factor to the difficulty of higher–order parity problems, but rather, more sophisticated evolutionary strategies are necessary.

5.2. *Scheduler*

The next experiment is a simple scheduler. A set of communication lines $\{\alpha_1, \alpha_2, ...\}$ are defined. The ordering of these lines, $\alpha_1 < \alpha_2 < \cdots$, denotes their scheduling order. Each communication line has a start and finish communication signal associated with it, designated by α_i and αf_i respectively. The rules for a scheduler with K lines are:

1. All actions must be seen once and only once.
2. The start signal α_i must be transmitted before any $\alpha_j (j > i)$ can be accepted.
3. After an α_i is received, the finish signal αf_i for that line must be eventually transmitted.

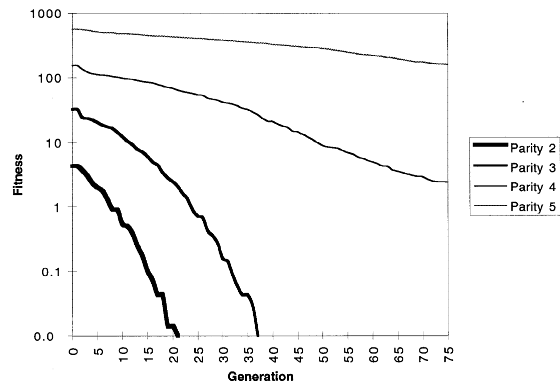

Fig. 8. Average best fitness, steady state system with tournament selection.

Table 5. Parameters: Scheduler-2

Parameter	Value
Functions	$+, \|, .$
Terminals	a, a', b, b'
Initial population size	750
Population size	500
Maximum generations	50
Probability replication	0.10
Probability crossover	0.90
Probability internal crossover	0.90
Max. depth initial population	6
Max. depth crossover offspring	17
Error tolerance	0.001
Trace limit	unique = 40, total = 90
Simplify offspring	off

5.2.1. Scheduler-2 Given the above specification, the scheduler-2 parameters are set as in Table 5. The actions a and af are the start and finish signals for line a, and b and bf for line b, which is scheduled after a.

An intention here is to evolve a program that exploits the interleaving operator "|". One challenge is to define an evaluation function that does not unduly penalize programs with interleaving. The evaluation strategy combines separate scores from the program's performance within a set of wrappers and the evaluation of the trace output from the program:

$$Score \; = \; \begin{cases} 500 \quad (if \;\; trace \;\; overflow) \\ \\ Score_1 + Score_2 \end{cases}$$

A program whose trace set was pre–empted due to overflow is penalized to ensure its demise.

The first score, the wrapper test, is computed as follows. The set of wrappers testing candidate program E are:

$$\begin{array}{c} (E \;\mid\; (\overline{a}.\overline{af}.\overline{b}.\overline{bf})) \setminus \{a, b, af, bf\} \\ (E \;\mid\; (\overline{a}.\overline{b}.\overline{af}.\overline{bf})) \setminus \{a, b, af, bf\} \\ (E \;\mid\; (\overline{a}.\overline{b}.\overline{bf}.\overline{af})) \setminus \{a, b, af, bf\} \end{array}$$

These three wrappers together completely define what we would like of our scheduler. Each case is a statement of observable behavior desired from the scheduler, and if correct should derive the stream

$$\ldots \;\; \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \mathbf{0}.$$

Internal communication within E is not possible with the given set of terminals. Should $\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \mathbf{0}$ be observed with the first wrapper expression, then the final bf action was not sent by E. Smaller output traces mean similarly deadlocked states were encountered.

After duplicate traces are filtered, the scoring of the trace results from the 3 wrapper tests is then:

$$Score_1 = (\sum(4 - |t_i|)) + (50 \times \#overflows)$$

t_i is an interpreted trace output of a wrapper, where $0 \leq |t_i| \leq 4$. Therefore, correct traces of size 4 do not increase the score. The first factor perfectly favors correct programs (Score = 0), while it most heavily penalizes programs that deadlock early. The latter factor penalizes wrapper tests that exceed their alloted trace overflow limits.

The wrapper tests give little information about erroneous internal behavior of expressions. Therefore, the program E is next interpreted by itself, and its trace set is scored. All the desired traces conforming to the scheduler specification are deleted, leaving a set of bad traces T of size N_{bad}. While doing this, the number N_{uncov} of desired traces *not* covered in T is determined. Next, each trace $t_i \in T_{bad}$ is evaluated for the following:

$M_i = \#$missing actions from $\{a,\; b,\; af,\; bf\}$
$R_i = \#$repeated actions
$O_i = \#$times where $a,\; b$ are out of order
$P_i = \#$instances a finish precedes its start (eg. af before a)

An overall score for all the bad traces for E is then determined:

$$Score_2 \; = \; \frac{\sum_{i=1}^{T_{bad}}(M_i + R_i + O_i + P_i)}{T_{bad}} + 3N_{uncov}$$

This computes the *average* bad score for the program. The reason for an average rather than a sum is because the latter would penalize programs having large trace sets resulting from "|". The number of traces N not covered by E is also tallied.

Runs with the above setup were usually successful. One run's fitness curve is in Figure 9, and the solution obtained in generation 11 is

$$a.((b.bf) \;\mid\; af)$$

This is the syntactically simplest CCS expression solving this problem, and makes optimal use of interleaving.

5.2.2. Scheduler–3 The extension of the scheduler problem to 3 lines significantly increased the complexity of search. The parameters are the same as Table 5, except terminals c and cf are added. The main change for the experiment was to relax the criteria for a solution. The following formula is used to combine the wrapper and badness scores in the evaluation function:

$$Score = \begin{cases} 5000 & (if\ trace\ overflow) \\ \frac{2*Score_1*Score_2}{Score_1+Score_2} \end{cases}$$

In this formula, the overall score becomes 0 if either the wrapper test or badness test are 0. Relaxation occurs because the wrapper test is less strict than the badness evaluation; after all, the universal interleaving expression will solve all the wrapper tests. Such degenerate solutions are avoided by using the trace limit parameters, since heavily interleaved expressions will be terminated by the interpreter. As before, the wrapper tests for desired behavior, while the badness tests give a direction to the search. The wrapper test, $Score_1$, will terminate evolution the moment that all the wrappers are satisfied. Solutions obtained may thus generate superfluous output, in addition to traces that solve the problem. It would not be difficult to write a postprocessor that takes a solution expression and removes extraneous terms that do not contribute to the solution trace set. Using the environment of the wrapper tests, for example,

$$(E \mid \overline{a}.\overline{b}.\overline{bf}.\overline{c}.\overline{af}.\overline{cf}) \setminus \{a,\ b,\ c,\ af,\ bf,\ cf\}$$

these erroneous traces are ignored via the restriction list.

One successful run's solution found the following at generation 29:

$$a.((af \mid (b.c.cf + a) + bf \mid bf) \mid bf)$$

This expression creates extraneous traces via "$bf \mid bf$", as well as a few with the final interleaved bf. However, all correct scheduling events are seen if it is used in an environment similar to the above. The fitness graph for this run is in Figure 10.

Note that the scheduler–3 problem is exactly and almost trivially solved if one exploits the scheduler–2 result within it. Using the terminal set $\{a, af, x\}$, a genetic programming run was done. The x action here is treated by the evaluator as a substitutive label for a solved scheduler–2 program over the actions $\{b, bf, c, cf\}$. This x is replaced by a scheduler–2 solution, during evaluation. With this strategy, most runs found a solution to scheduler–3 in the initial generation, as the solution is the simple expression:

$$a.(af \mid x)$$

Performing the substitution on this,

$$a.(af \mid b.((c.cf) \mid bf))$$

results in the optimal solution for scheduler–3. Higher scheduler–K problems scale upwards similarly.

After seeing the above optimal solutions for scheduler–2 and –3, one might question the inclusion of "+" as a nonterminal in those experiments. The reason is that, without it, the *only* solution possible is the optimal solution: the search must converge to precisely one single program, which is unreasonable. With the

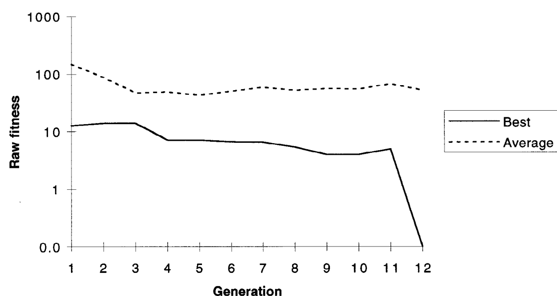

Fig. 9. Fitness curve, Scheduler–2.

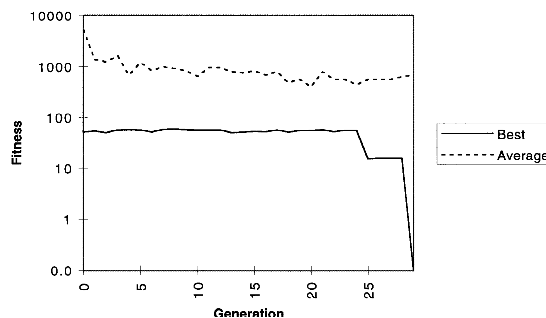

Fig. 10. Fitness curve, Scheduler–3.

inclusion of "+", a variety of solutions are possible, and a solution is more likely.

5.3. System implementation

The entire CCS genetic programming system is implemented in Quintus Prolog 3.2 on a Silicon Graphics 150MHz R4400 system. A typical iteration of a single scheduler–3 population with a population of 500 and trace limits set to $(unique = 40,\ all = 90)$ takes approximately 40 seconds. This includes 16 separate calls to the CCS interpreter per individual, in addition to trace analyses. The parity–3 runs take approximately 15 seconds per generation.

6. Conclusion

The main challenge in applying genetic algorithms towards concurrent computations is the vast execution space encountered with nondeterminism. If appropriate user–defined controls are placed upon the execution of the CCS interpreter, many concurrent problems are indeed evolvable. This of course places constraints on the genetic diversity of the population, since a heavily interleaved program may have many desirable characteristics, yet may be discarded due to trace overflow. In addition, CCS is a lower–level language than Lisp, and the styles of problems solved here are more challenging to realize in CCS. For example, the use of sequences of input signals for the even–parity functions is a significant problem specification, since the actual communication protocol must be evolved from scratch. This is circumvented in a Lisp program which can access parameters from the function header, and apply boolean primitives to them directly.

While the experiments presented were selected for their successful solution to the problems in question, admittedly many unsuccessful runs were encountered. One improvement would have been the use of tournament selection, which would require less *ad hoc* evaluation functions. Fitness–proportional selection schemes as used here often requires numerical massaging of the fitness functions to create a suitable search terrain, which tournament selection does not need.

This research only addressed non–iterative concurrent problems, and the genetic programming of iterative and recursive concurrent systems is currently being studied. One promising avenue is to look closely at Koza's ADF system, since concurrent programs are modular in nature. Enhanced evolutionary strategies such as coevolution and Lamarckism may be useful in the difficult search space of concurrent programs [13]. Other process algebra could be studied, each of which may present particular technical challenges for GP.

Applications of other machine learning paradigms to concurrency have been attempted. [11, 12] discusses the derivation of process algebraic expressions using computational learning algorithms. This approach is hampered by complexity issues, and requires severe restrictions on the scope of the process algebraic expressions handled. Genetic programming is much more promising in this regard, because the success of program evolution is entirely dependent upon the existence of suitable evaluation functions, and does not require unreasonable restrictions on the target language. Work in deriving concurrent genetic programming environments, for example [6], should not be confused with this paper's the genetic evolution of concurrent programs.

References

1. I.K. Aalbersberg and G. Rozenberg. Theory of Traces. *Theoretical Computer Science*, 60:1–82, 1988.
2. Wilker Shane Bruce. *The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs*. PhD thesis, School of Computer and Information Sciences, Nova Southeastern University, December 1995.
3. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
4. E.C.R. Hehner and A.J. Malton. Termination Conventions and Comparative Semantics. *Acta Informatica*, 25:1–14, 1988.
5. J.R. Koza. *Genetic Programming*. MIT Press, 1992.
6. S.R. Maxwell. Experiments with a Coroutine Execution Model for Genetic Programming. In *Proceedings 1st IEEE Conference on Evolutionary Computation*, pages 413–417, 1994.
7. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
8. P. Nordin. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In K.E. Kinnear, editor, *Advances in Genetic Programming*, pages 311–331. MIT Press, 1994.
9. E.-R. Olderog and C.A.R. Hoare. Specification-Oriented Semantics for Communicating Processes. *Acta Informatica*, 23:9–66, 1986.
10. J.L. Peterson. Petri Nets. *Computing Surveys*, 9(3), September 1977.
11. B.J. Ross. The Inductive Inference of Cyclic Synchronized Interleaving. In *Proceedings of the 11th European Conference on Artificial Intelligence*, Amsterdam, 1994. John Wiley and Sons.
12. B.J. Ross. PAC Learning of Interleaved Melodies. In *1995 IJCAI Workshop on Music and Artificial Intelligence*, pages 96–100, Montreal, Quebec, 1995.

13. B.J. Ross. A Lamarckian Evolution Strategy for Genetic Algorithms. In L. Chambers, editor, *The Practical Handbook of Genetic Algorithms*, volume 3. CRC in Press.
14. M.L. Wong and K.S. Leung. Learning Programs in Different Paradigms using Genetic Programming. In *Proceedings 4th Congress of the Italian Association for AI*, pages 353–364, 1995.

Brian Ross received a BSc from the University of Manitoba, MSc from the University of British Columbia, and PhD from the University of Edinburgh. Currently he is an associate professor of computer science at Brock University. His research interests include evolutionary algorithms, concurrency, logic programming, and computer music.