USING GENETIC PROGRAMMING TO SYNTHESIZE MONOTONIC STOCHASTIC PROCESSES†

Brian J. Ross
Department of Computer Science
Brock University
500 Glenridge Avenue
St. Catharines, Ontario, Canada L2S 3A1
email: bross@brocku.ca

ABSTRACT
The automatic synthesis of stochastic concurrent processes is investigated. We use genetic programming to automatically evolve a set of stochastic π-calculus expressions that generate execution behaviour conforming to some supplied target behaviour. We model the stochastic π-calculus in a grammatically-guided genetic programming system, and we use an efficient interpreter based on the SPIM abstract machine model by Phillips and Cardelli. The behaviours of target systems are modelled as streams of numerical time series for different variables of interest. We were able to successfully evolve stochastic π-calculus systems that exhibited the target behaviors. Successful experiments considered target processes with continuous monotonic behaviours.

KEY WORDS
Genetic programming, process algebra, dynamic systems.

† To be presented at *Computational Intelligence 2007*, Banff, AB, Canada, July 2-4, 2007.

1 Introduction

Process algebra are formal systems used to study the behaviour of concurrent systems [1][2]. One such process algebra is the stochastic π-calculus [3]. It has simple and elegant syntax and semantics, and like other process algebras, it is useful as a specification language for concurrent systems. Lately, researchers have applied the stochastic π-calculus towards the modeling of various chemical and biological networks [4] [5] [6]. A modeled system can be interpreted, which means simulations can be performed. The behaviour of these simulations is characterized by time-course data, in which symbolic quantities of elements of the system change over time. Hence, a system is characterizable by sets of time-series plots of variable quantities.

Given one or more instances of such time-series data, however, it is difficult to derive a stochastic π-calculus expression which might express such behaviour. Although the semantics of the stochastic π-calculus language is straightforward, it is a nondeterministic, probabilistic algebra, and processes created from it belong to the family of dynamic systems [7]. The behaviours of dynamic systems are notoriously difficult to predict, as long-term behaviour is a result of complex and intricate nondeterministic interactions of system components over the course of time. It is a challenging task for someone to write a stochastic π-calculus system that could generate a given sample of time-course data for variables.

We use genetic programming (GP) [8][9] to evolve stochastic π-calculus processes. The target behaviour for the evolved systems take the form of time-course data, in which different variable quantities change over time. The process algebra is denoted by a context-free grammar. One major purpose for using a grammar within the GP system is to help constrain the forms of expressions derivable within it, by pre-specifying more pragmatic general forms of expressions. Without such grammatical constraints, evolution can be burdened by nonsensical and inefficient candidate expressions.

Section 2 reviews the stochastic π-calculus, machine learning and dynamic systems, and genetic programming. Details about two experiments for synthesizing a stochastic concurrent processes are given in Section 3. Results of the experiments are reported in Section 4. Conclusions and future work are discussed in Section 5.

2 Background

2.1 The stochastic π-calculus

Process algebra are formal systems used to study concurrency [1][2]. The stochastic π-calculus is a process algebra used to model stochastic processes that have dynamic structural behaviour [3]. A number of implementations of the calculus are available [10] [11] [12]. For a complete discussion of the semantics of the stochastic π-calculus see [3] [12].

The syntax for the subset of the stochastic π-calculus used in this paper is:

$$
\begin{aligned}
P &::= \mathbf{0} \ \mid \ P\#P \ \mid !proc : P \ \mid \ \Sigma \\
\Sigma &::= \pi : P \ \mid \ \Sigma + \Sigma \\
\pi &::= c \ \mid \ \overline{c} \ \mid \ \overline{proc}
\end{aligned}
$$

A process P is either a null process $\mathbf{0}$, two processes running in parallel via $\#$, a process definition defined with a replication operator !, or a choice expression Σ. A choice expression is either one or more terms that can be stochastically selected. A term is either an input (x) or output $(\overline{x})$ action, or process call. Process calls are output actions with process names.

The essential idea of process execution in the (non-stochastic) π-calculus is the following transition:

$$(x : P_1 + \Sigma_1)\#(\overline{x} : P_2 + \Sigma_2)\#P_3 \quad \overset{rate(x)}{\rightarrow} \quad P_1\#P_2\#P_3$$

Here, a synchronous handshaking communication has arisen along channel x, and the entire expression has transformed. The other choices of actions in the Σ_i terms have been pre-empted by this communication.

The stochastic π-calculus enhances this execution model by considering communication rates of channels and the quantities of channels available for communication. When a number of synchronous communications like x above are available, the semantics will select a particular action based upon its probabilistic likelihood relative to other actions available. An action that has a higher rate of execution or more instances available in the overall expression, will be more likely to be selected for a transition. This stochastic selection mechanism is called the *Gillespie algorithm*, and was invented for use in chemical reaction simulations [13]. It first computes the overall probabilities of the active channels (those with both output and input terms connected via parallel composition), and selects one via an exponential distribution. A result of the selection will be a transformation of the overall expression as above, as well as an update to a global time variable. The time is updated by an amount inversely proportional to the action probability, which reflects the higher frequency of more probable actions.

Using the above ideas, a stochastic π-calculus simulation will involve the dynamic alteration of an expression, whose instances (quantities) of various channels within terms change over time. Hence the behaviour of the process can be described by time-course plots of the channels. Furthermore, the nondeterministic nature of the simulation means that repeated simulations will usually result in different time-course plots, reflecting the stochastic nature of the process.

2.2 Dynamic systems and their inference

Dynamic systems are systems that change state over time [7]. Many natural phenomena are modeled by dynamic systems. For example, a swing in motion can be modeled by its position in space over time. The metabolic chemical reactions within a cell can be modeled by the quantities of interacting proteins and substrates over time. The regulation of networks of interacting genes can be modeled by the expression and inhibition rates of the genes over time. The execution of stochastic processes as modeled by the stochastic π-calculus is another another instance of a dynamic system.

The complex and chaotic nature of many dynamic systems has attracted the attention of the machine learning community. Dynamic systems are difficult for human beings to conceive and perceive. On the other hand, perhaps they may be conducive for automatic analysis by computers. Examples of the use of neural networks and/or evolutionary computation to infer dynamic systems and time series include [14] [15] [16] [17] [18] [19] [20] [21].

2.3 Genetic programming

Genetic programming (GP) is a an evolutionary algorithm, in which individuals in the population denote computer programs [8] [9]. Chromosomes typically take the form of parse trees, in which internal nodes are functions, branches are argument expressions, and leafs are constants. Reproduction operators manipulate this tree structure, and must ensure that resulting trees are syntactically correct. In addition, a *closure* property specifies that all operators in the language must execute correctly on any argument value. This is necessary because randomly constructed and manipulated trees may result in many unforeseen language structures.

In the original implementiation of GP, the tree encoded a Lisp S-expression, which could therefore be directly executed by the Lisp interpreter. Subsequent implementations have experimented with a variety of representations and target languages. Grammatically-guided GP is used in this paper [22][23]. Here, the tree denotes a parse tree as represented by a context-free grammar, and tree creation and reproduction must respect grammatical correctness as defined by this grammar. An advantage of grammatically-guided GP is that the search space of program structures can be constrained by the grammars. This permits more sensible program structures to be constructed, which may result in higher-quality and more optimal solutions. Another advantage is that the closure property is easier to implement.

3 Experiments

3.1 An interpreter for the stochastic π-calculus

It is important that the execution of stochastic π-calculus expressions is as efficient as possible. The genetic programming system will need to evaluate thousands of candidate expressions during the course of a run. Furthermore, due to the stochastic nature of the processes, each expression will need to be interpreted multiple times, in order to determine an average behaviour to evaluate.

A stochastic π-calculus interpreter has been implemented in Sicstus Prolog [24]. This interpreter is based on a stochastic π-calculus abstract machine (SPIM) [12]. The SPIM specification is the basis for an interpreter they wrote in OCAML, a symbolic object-oriented functional

$$
\begin{array}{rcl}
\mathit{Start} & ::= & \mathit{ChRates},\ \mathit{Proc}\#\mathit{Proc}\#\mathit{Topexpr} \\
\mathit{ChRates} & ::= & \mathit{Float},\ \mathit{Float},\ ... \\
\mathit{Proc} & ::= & !\mathbf{proc}_i : \mathit{Expr} \\
\mathit{Topexpr} & ::= & \mathit{Int}@(\mathit{PiCall} : \mathit{Guardseq}) \\
 & & \mid\ \mathit{Topexpr}\#\mathit{Topexpr} \\
\mathit{Expr} & ::= & \mathit{Choice}\ \mid\ \mathit{Expr}\#\mathit{Expr} \\
\mathit{Choice} & ::= & \mathit{Pi} : \mathit{Guardseq}\ \mid\ \mathit{Choice} + \mathit{Choice} \\
\mathit{Guardseq} & ::= & \mathbf{0}\ \mid\ \mathit{PiCall} : \mathit{Guardseq}\ \mid\ \mathit{Expr} \\
\mathit{PiCall} & ::= & \mathit{Pi}\ \mid\ \overline{\mathbf{proc}}\ \mathit{Int} \\
\mathit{Pi} & ::= & \mathit{NormCh}\ \mid\ \overline{\mathit{NormCh}} \\
\mathit{NormCh} & ::= & \mathbf{c} \in \text{channels} \\
\mathit{Float} & ::= & \mathit{min}_f \leq \mathbf{f} \leq \mathit{max}_f \\
\mathit{Int} & ::= & \mathit{min}_i \leq \mathbf{i} \leq \mathit{max}_i
\end{array}
$$

Figure 1. A grammar for the stochastic π-calculus

language. Converting SPIM into a logic program based interpreter is similarly straightforward.

The stochastic π-calculus semantics are simple enough that the semantic rules can be converted virtually directly into statements of a symbolic language such as OCAML or Prolog. Unfortunately, the resulting execution will not be efficient. This is due to the fact that sound probabilistic evaluation of expressions requires that the entire set of π-calculus terms be reduced into a canonical form consisting of basic summation terms. These summations are then enumerated to determine the probabilities used by the Gillespie algorithm. The resulting probabilities determine which action pair is chosen for execution. This reduction and enumeration process is done each time a communication reduction occurs. The net result is that the reductions will be repeatedly performed during each state transition of the system, and most often are wastefully duplicated during each transition.

The essential idea of the abstract machine description is to convert high-level π-calculus terms into an intermediate representation that is more efficient to process, because it does not require repeated reduction during every transition of the system. The abstract machine representation saves earlier reductions and enumerations of expressions, and hence prevents their re-reduction during every transition. Only the terms that were directly involved in a transition are further reduced, and any changes due to these reductions are efficiently applied to the overall probabilities for the entire system. The net effect is a stochastic π-calculus interpreter that is much more efficient than one that works directly from raw expressions.

Although our interpreter handles the complete stochastic π-calculus, channel passing and restriction are not treated here, and process definitions are implemented without parameter passing [3].

3.2 A grammar for the stochastic π-calculus

We use grammatically guided genetic programming to evolve process algebra expressions. Our main intention for using grammatical GP is to constrain the possible forms of stochastic π-calculus expressions evolved. Without sensible grammatical constraints, it is too easy for the GP system to evolve large, complex, and inefficient expressions. The DCTG-GP system is used in our experiments [23]. This is a Prolog-based system, and it uses a definite-clause translation grammar (DCTG) to specify the target language to be evolved by genetic programming. A DCTG is a logical implementation of a context-free attribute grammar, and it permits the syntax and semantics of the GP language to be defined together.

A Backus-Naur Form (BNF) grammar for the stochastic π-calculus is in Figure 1. In the figure, capitalized labels are nonterminals, and lower case or bold-face are terminals. This grammar permits the channel rates to be evolved with an expression, as denoted by the *chRates* field. This will contain one floating point value per channel (unless overridden by the user). The *Start* rule defines expressions for each labelled process for the target expressions. Two references to *Proc* here will mean that two process expressions will be required. The *Topexpr* rule permits replicated expressions; for example, $5@E$ means that 5 parallel instances of expression E are created. Furthermore, each such expression is guarded by either a basic communication (*Pi*) or process call. The *Expr*, *Choice*, and *Guardseq* rules define the composition of π-calculus expressions. One constraint introduced in these rules is the use of *Pi* term guards in choice expressions, which ensures that each choice term introduces a communication, rather than a process call. Process calls are determined by an integer field, which when evaluated *modulo K* will specify a call to one of K processes.

3.3 Evaluation of stochastic trajectories

The stochastic processes being studied are dynamic systems with behaviours that change nondeterministically over time. This is in contrast to deterministic systems, whose behaviour is uniquely characterized by the conditions set during the starting state. Stochastic processes are challenging to analyze in a machine learning environment. Typically, systems to be learnt are not precisely characterizable by single time series of values. Rather, a target behaviour might be described by an average time series, possibly supplemented with statistical descriptions such as standard deviation. Likewise, candidate processes being evaluated during a GP run are nondeterministic, and a system can vary substantially during different interpretations.

The target system is described by a single set of time series, where each component series is the numerical state change of some variable of interest. Let A be a variable of a probabilistic process. Then trajectory j for this variable

```
Score $\leftarrow 0$
$S \leftarrow$ interpret(E, max time, max transitions)
For each variable $A$ with averaged target series $\overline{A}$:
    Let $A'$ be the observed behaviour of $A$ in $S$.
    For each $(t_i, a_i) \in \overline{A}$:
        If ($t_i >$ max time in $A'$) and $(a_i > 0)$
            Score $\leftarrow$ Score+*Penalty*
        else
            $\tilde{a} \leftarrow$ linear interpolate($A'$, $t_i$)
            Score $\leftarrow$ Score $+ \mid a_i - \tilde{a} \mid$
```

Figure 2. Evaluation of time series fit for process E

is denoted by the sequence,

$$A^j = (t_i^j, a_i^j) \qquad i = 1, ..., k^j$$

where $a_i^j \geq 0$ is the value of variable A at time t_i^j at transition step i, and the maximum transition step for this trajectory is k^j. In the case of probabilistic target systems, an average time series $\overline{A}$ is computed from a set of trajectories A^j.

Stochastic processes are evaluated as follows. An expression E is interpreted, resulting in a set S of time series composed of individual series for different variables within E. The interpreter is supplied with maximum time and transition count limits, which will pre-empt the interpreter when necessary. There is a set of target variables of interest, whose behaviour is represented by the (average) time series described above. Should a variable of interest be missing from E, its time series will be empty. The sum of absolute errors is then computed for each variable series (Figure 2). If the interpreted series ends prematurely or is empty, then a penalty is administered when a non-zero target value is expected. Otherwise, the absolute error is computed. Because the target times do not necessarily match those of the interpreted series, the latter must have values linearly interpolated at the given target times. The absolute difference between the target and interpolated value at time t_i is then tallied for the entire target series. Performing the above for the entire interpreted series will result in an overall error ($Score$). The goal is to minimize this score.

The above evaluates one set of variable trajectories for a process expression E. This scoring is then done on a given number of separate interpretations and resulting trajectories, and the mean of all these scores is used as the overall fitness value for the process.

3.4 Special reproduction operators

Standard crossover and mutation operators for grammatical GP are used [23]. To help in the evolution of processes with active communication behaviours, some specialized reproduction operators are introduced.

Single rebalance mutation: All the input channels, output channels, and process calls in an expression are tallied. The least referenced channel or process is determined, and a term is constructed for it. For example, if there are no output calls to channel a, then one is created, and it replaces a random term in the expression.

Complete rebalance mutation: Every channel call missing an input or output reference has a term created for it, and it is placed in the expression. Missing processes calls are also placed in the expression.

Note that the random placement of these constructed terms may create new channel and process imbalances. For expediency's sake, these are not remedied.

3.5 GP parameters

Parameter	*Value*
Pre-culled population	2000
Population	1500
Generations	70
Runs	20
Initialization	ramped half&half
Initial max tree depth	8
Max tree depth	12
Int. crossover	0.77
Ext. crossover	0.13
Terminal mutation	0.05
Internal mutation	0.02
Single rebal. mut.	0.02
Complete rebal. mut.	0.01
Tournament size	4
Elite migration	5
Penalty value	600
Lamarckian evol.	33% of init popn, 5% of popn, every 7 gen.

Table 1. Common GP parameters

Parameters common to both experiments are given in Table 1. Many parameters are standard in the literature [9], and are not discussed further. A few unique details are as follows. The initial population of 2000 is culled to 1500, by removing the weakest expressions. This culled population is then given a “Lamarckian boost” by performing local search on the weakest 33% of the individuals. Reproduction operations are applied to each of these individuals, until either an improvement of fitness is obtained, or a maximum of 5 operations are attempted. The improved expression, if any, then replaces the original one in the population. Lamarckian evolution is also applied to the population periodically during the run, using the approach in [25]. Here, 5% of the population is selected with tournament selection, and 6 reproduction operations are performed on each individual. The best performing altered individual replaces the original.

4 Results

4.1 Experiment 1: Two-variable system

Parameter	*Value*
Targets	poly dec $\leftrightarrow proc_0$, ramp inc $\leftrightarrow proc_1$
Channels	a, b
Channel rates	evolved
Processes	proc0, proc1
Time limit	10.0
Transition limit	2000
Float range	$0.75 \leq f \leq 12.0$
Integer range	$2 \leq i \leq 110$
Target series size	101
# interpretations/eval	3

Table 2. Experiment 1 parameters

	Train	Test
Average score	634	912
Std dev	113	200
Best score	494	676
Worst score	890	1315
Confidence range, low:high (95%)	581:687	818:1006
Mean error per example	3.14	4.51
Correlation train & test	0.82	

Table 3. Summary of results: solution evaluation

This experiment uses two target sequences with simple monotonic shapes – a polynomially decreasing sequence, and a ramped increasing sequence (see Figures 3 and 4). Other parameters are given in Table 2. All channel rates (a, b, both processes) are evolved.

Table 3 summarizes the results for the 2-variable runs. The mean solution score is within the 95% confidence range. By dividing the scores by the number of variables (2) and sample points (101), the overall error per sample point is about 3. There was a high correlation between the fitness scores and testing scores, which shows that the averaging of fitness scores is good for obtaining predictable processes.

The plots in Figures 3 and 4 show the target plot, the (averaged) output of all the solutions from the 20 runs, and the output from the solution with the best fitness score. Note that a run's best solution is the expression which obtained the best fitness (lowest accumulative error), based on the average of 3 separate interpretations and their corresponding fitness evaluations. Since these are probabilistic processes, however, subsequent interpretations are not guaranteed to exhibit excellent performance. The plots

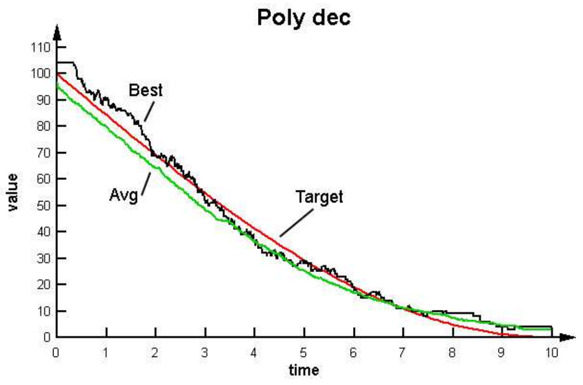

Figure 3. Polynomial decreasing: target, run avg, and best

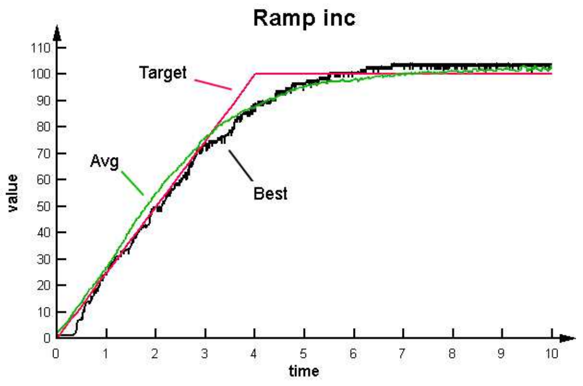

Figure 4. Ramped increasing: target, run avg, and best

show that all the solutions as plotted by the average exhibit a reasonable fit to the target behaviour. The expression for the single solution plotted in the above figures is:

$$
\begin{array}{l}
!proc_0 : (\overline{b} : 0) \;\# \\
!proc_1 : ((\overline{b} : \overline{b} : 0)\#(b : \overline{proc_1} : 0)) \;\# \\
103@(\overline{proc_0} : \overline{proc_0} : \overline{proc_1} : \overline{proc_0} : 0) \;\# \\
34@(\overline{b} : 0) \;\# \\
37@(\overline{a} : 0) \\
rates: \quad a = 0.88 \qquad proc_0 = 0.86 \\
\qquad\quad\; b = 1.00 \qquad proc_1 = 0.89
\end{array}
$$

4.2 Experiment 2: Ethylene

Here, we evolve a process which generates behaviour that matches that of a simulation for ethylene given in [26]. The SPIM expression was translated into the following:

$$
\begin{array}{l}
!proc_0 : (ay : \overline{proc_1} : 0 + ar : \overline{proc_1} : 0) \;\# \\
!proc_1 : (\overline{ar} : \overline{ep} : 0 + \overline{pr} : 0 + pr : 0) \;\# \\
4@(\overline{ay} : 0) \;\# \\
200@(\overline{proc_0} : 0)
\end{array}
$$

Ten interpretations were performed, and an average behaviour was computed for use as the target behaviour.

Parameter	*Value*
Targets	4 variables from Ethylene simulation
Channels	ay, ep, pr, ar
Channel rates	fixed: ay=1.0, pr=1.0,ar=10.0, default=100000000.0
Processes	$proc_0$, $proc_1$
Time limit	0.20
Transition limit	2000
Float range	$0.75 \leq f \leq 12.0$
Integer range	$1 \leq i \leq 200$
Target series size	145
# interpretations/eval	3

Table 4. Experiment 2 parameters

	Target	Train	Test
Avg score	4925	8808	9724
Std dev	1728	2441	2459
Best score	2436	3639	4149
Worst score	8636	11898	12712
Confidence range, low:high (95%)	4117: 5733	7666: 9950	8573: 10875
Mean error per example	8.5	15.1	16.8
Correlation train & test		0.97	

Table 5. Summary of results: solution evaluation

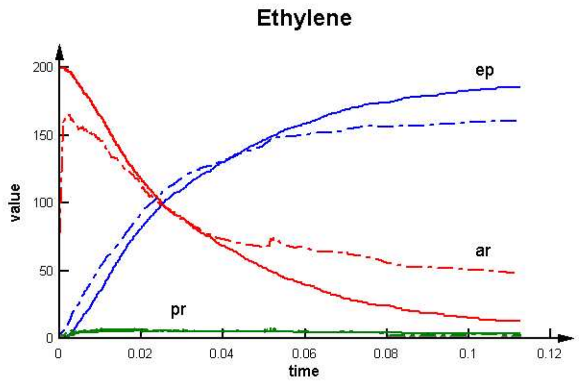

Figure 5. Ethylene: target (solid) and avg solution (dash)

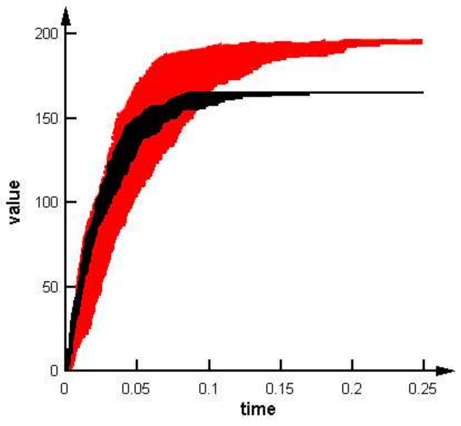

Figure 6. Variable EP range from 10 interpretations: target (red) and best (black)

Parameters for this experiment are in Table 4. Although the target system uses 4 channels and 2 subprocesses, we only match the 4 channels during fitness evaluation. We also supply all channel rates, to match those of the target system. Note that this increases the problem difficulty for evolution, since predetermined channel rates are an additional problem constraint.

Table 5 summarizes the results for the 20 runs. The first column shows how the above solution expression measures against itself, by evaluating its output from 20 separate simulations against the average plot used as training data. Looking at the evolved solutions, there is a substantial gap between the scores of the best and worst solutions from the 20 runs. Statistically, 95% of solutions obtained will generate scores within the confidence range specified. The test scores are single interpretations of the designated solution from each run. Test scores tend to be weaker than the fitness scores. There is a very positive correlation (0.97) between the fitness and test scores.

Plots of the average solution behaviour from 20 runs against the target behaviour are in Figure 5. This plot shows 3 channels, since the ar and ay plots are almost identical. The curves for this solution and others tend to be more accurate during the early portion of the execution, due to a bias introduced in our time-series evaluation strategy. The source expression's interpretation produces more frequent but smaller-duration time changes early in the execution: the time up to 0.04 seconds contained 117 of the data points, while the remaining time span up to 0.10 seconds used 28 points. Hence solutions tended to be more divergent after 0.04 seconds, since scoring was not as precise.

One solution expression is:

$$\begin{array}{l}
!proc_0 : (\overline{ay} : ep : \overline{proc_0} : \overline{pr} : \overline{proc_0} : 0 + ar : ep : 0)\ \# \\
!proc_1 : (ar : ep : \overline{ar} : \overline{proc_0} : 0 + ay : \overline{proc_0} : 0)\ \# \\
39@(\overline{proc_1} : \overline{proc_0} : 0)\ \# \\
5@(\overline{pr} : 0)\ \# \\
72@(\overline{proc_1} : 0)\ \# \\
27@(\overline{proc_1} : \overline{proc_1} : 0)
\end{array}$$

Note that this expression is not similar to the source expression. This is because the target behaviour of the source expression is underspecified. Without the introduction of more constraints to help refine the characteristics of the target system, many evolved expressions can be reasonable candidate solutions. For example, for more accurate process inference, the behaviours of the source expression processes should also be included in the fitness evaluation.

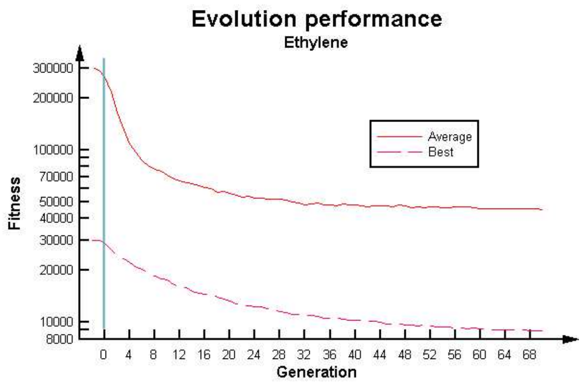

Figure 7. Performance graph: avg of 20 runs

Figure 6 shows the variation of one variable (EP) that occurs between separate interpretations of the same expression. The red portion shows the range seen from 10 interpretations of the target expression, and the black is the range from the above evolved solution. This particular solution underestimates the final EP value.

Figure 7 shows the average population fitness and best of generation fitness, averaged over the 20 runs. Due to the fitness evaluation's use of penalty scores, the population average is considerably higher than the best individual, and hence a logarithmic scale is used on the Y-axis. The curves to the left of the vertical line at generation 0 shows the effect of the culling and Lamarckian boost of the initial population.

5 Conclusion

This paper described two example experiments in which genetic programming is used to automatically synthesize stochastic processes from example time series of their desired behaviours. Many other experiments were performed in the course of this research. We were successful in evolving processes for monotonic, continuous time series. Furthermore, we used up to 4 variables during the evaluation. Naturally, systems with more variables to infer are more difficult to evolve.

There are a number of new directions under investigation for this research. Experiments that used cyclic target behaviours were unsuccessful. When trying to evolve a process that matches a sine curve output, GP would evolve processes with flat output midway through the sine curve. The difficulty with cyclic and nonmonotonic behaviour is due to a number of factors. Firstly, cyclic behaviours are particularly difficult to evaluate using the simple evaluation strategy used in this paper. Cyclic stochastic processes tend to exhibit a much higher deviation of behaviour than monotonic processes, and this does not bode well with *sum of absolute errors* fitness evaluations. Secondly, sophisticated stochastic π-calculus expression structures are required for creating cyclic, non-monotonic behaviours, and channel passing is usually necessary for such systems. Although our implementation supports channel passing, we found that such expressions are too brittle for the reproduction and grammatical definition used in this paper. Evolution in our experiments spent much time removing deadlock from expressions, and so the additional intricacies of channel passing are too difficult to handle. Cellular encoding could be a more fruitful representation for such structures (see below).

There are other examples of research that use evolutionary computation to evolve network with time-series behaviour characteristics. Genetic algorithms have been used to evolve stochastic Petri systems for modelling metabolic networks [27]. They successfully evolve a Petri net that models a phospholipid pathway, from time-course data for the variables (substrates) of the target system. Although their evaluation strategy is similar to ours, their encoded representation within the GA is considerably different. The linear decoding of genes automatically results in a connected, non-deadlocked network. Our GP system, on the other hand, can often produce deadlocked, disconnected π-calculus expressions. This does inhibit the performance of evolution.

Koza *et al.* use GP to evolve metabolic networks [28]. Their target network structures are similar in flavour to ours, in that the network execution is characterizable by time-course data for the variables of interest. However, like [27], their networks are deterministic, and network behaviour depends solely upon starting conditions (values) of the network. Furthermore, their GP representation uses cellular encoding, in which execution of a GP tree will refine an initial embryo network into a more complex one. This strategy is an ideal one for network-like structures, and would circumvent many of the deadlock issues we encountered.

This research is another step in an ongoing plan for using GP to evolve more complex and generalized processes. Earlier research by the author investigated the use of GP to evolve deterministic processes [29] and cyclic processes [30]. A goal is to automatically evolve processes that model chemical and biological networks [4] [5] [6].

Acknowledgment: Thanks to Janine Imada for helpful comments. Supported by NSERC Operating Grant 138467.

References

[1] R. Milner, *Communication and concurrency* (New York: Prentice Hall, 1989).

[2] R. Milner, *Communicating and mobile systems: the pi-calculus* (Cambridge, UK: Cambridge University Press, 1999).

[3] C. Priami, Stochastic pi-calculus, *The Computer Journal*, *38*(7), 1995, 579–589.

[4] R. Blossey, L. Cardelli, & A. Phillips, A compositional approach to the stochastic dynamics of gene networks, *Trans. in Comp. Sys. Bio (TCSB)*, *3939*, 2006, 99–122.

[5] A. Phillips, L. Cardelli, & G. Castagna, A graphical representation for biological processes in the stochastic pi-calculus, *Trans. in Comp. Sys. Bio (TCSB)*, *4230*, 2006, 123–152.

[6] C. Priami, A. Regev, E. Shapiro, & W. Silverman, Application of a stochastic name-passing calculus to representation and simulation of molecular processes, *Information Processing Letters*, *80*, 2001, 25–31.

[7] S.H. Strogatz, *Nonlinear dynamics and chaos* (Cambridge, MA: Westview Press, 1994).

[8] W. Banzhaf, P. Nordin, R.E. Keller, & F.D. Francone, *Genetic programming: An Introduction* (San Francisco: Morgan Kaufmann, 1998).

[9] J.R. Koza, *Genetic programming: On the programming of computers by means of natural selection* (Cambridge, MA: MIT Press, 1992).

[10] BioSPI, The biospi project, http: // www.wisdom.weizmann.ac.il ~ biospi/, last accessed March 1, 2007.

[11] A. Bloch, B. Haagensen, M.K. Hoyer, & S.U Knudsen, The stopi-calculus and simulator, 2004, last accessed March 1, 2007.

[12] A. Phillips & L. Cardelli, A correct abstract machine for the stochastic pi-calculus, in *Proc. Bioconcur'04*, London, UK, 2004.

[13] D.T. Gillespie, Exact stochastic simulation of coupled chemical reactions, *J. Phys. Chem*, *81*, 1977, 2340–2361.

[14] P.J. Angeline, Evolving predictors for chaotic time series, in *Proc. SPIE: Application and Science of Computational Intelligence*, *3390*, 1998, 170–180.

[15] R. Bakker, J.C. Schouten, C.L. Giles, F. Takens, & C.M. van den Bleek, Learning chaotic attractors by neural networks, *Neural Computation*, *12*, 2000, 2355–2383.

[16] H. Cao, L. Kang, & Y. Chen, Evolutionary modeling of ordinary differential equations for dynamic systems, in *Proc. GECCO 99*, Orlando, FL, 1999, 959–965.

[17] B. Grosman & D.R. Lewin, Automated nonlinear model predictive control using genetic programming, *Computers and Chemical Engineering*, *26*, 2002, 631–640.

[18] M.A. Kaboudan, Forecasting with computer-evolved model specifications: a genetic programming application, *Computers and Operations Research*, *30*, 2003, 1661–1681.

[19] G.Y. Lee, Time series perturbation by genetic programming, in *Proc. CEC 2001*, Seoul, Korea, 2001, 403–409.

[20] M. Witczak, A. Obuchowicz, & J. Korbicz, Genetic programming based approaches to identification and fault diagnosis of non-linear dynamic systems, *Int. J. Control*, *75*(13), 2002, 1012–1031.

[21] W. Zhang, G. Yang, & Z.Wu, Genetic programming-based modeling on chaotic time series, in *Proc. 3rd Intl Conf. on Machine Learning and Cybernetics*, Shanghai, China, 2004, 2347–2352.

[22] R.I. McKay, X.H. Nguyen, P.A. Whigham, & Y. Shan, Grammars in genetic programming: A brief review, in *Proc. Intl. Symp. on Intelligence, Computation and Applications*, Wuhan, China, 2005, 3–18.

[23] B.J. Ross, Logic-based genetic programming with definite clause translation grammars, *New Generation Computing*, *19*(4), 2001, 313–337.

[24] SICS, *SICStus prolog user's manual*, May 2005, http://www.sics.se/isl/sicstus.html.

[25] B.J. Ross, A Lamarckian evolution strategy for genetic algorithms, in *The Practical Handbook of Genetic Algorithms*, L. Chambers, Ed., *3*, (Boca Raton: CRC Press, 1997), 1–16..

[26] A. Phillips, The stochastic pi machine (spim), http://research.microsoft.com/ aphillip/spim/. Last accessed March 1, 2007.

[27] J. Kitagawa & H. Iba, Identifying metabolic pathways and gene regulation networks with evolutionary algorithms, in *Evolutionary Computation in Bioinformatics*, G.F. Fogel and D.W. Corne, Eds., (San Francisco: Morgan Kaufman, 2003), 255–278.

[28] J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, & G. Lanza, *Genetic programming IV: routine human-competitive machine intelligence*, (Norwell, MA: Kluwer Academic Publishers, 2003).

[29] B.J. Ross, The evolution of concurrent programs, *Applied Intelligence*, *8*(1), January, 1998, 21–32.

[30] B.J. Ross, Pairwise sequence comparison and the genetic programming of iterative concurrent programs, in *Proc. Genetic Programming 1998*, Madison, WI, 1998, 338–343.