An Optimised Genetic Algorithm for Competitive Robocode Agents

Zak Edwards

Z.Edwards@warwick.ac.uk

University of Warwick

Department of Computer Science

May 5, 2016

Module Organiser:	Prof. Nathan Griffiths
Head of Department:	Prof. Stephen Jarvis

Contents

Nomenclature

Unless otherwise defined.

G	We define an **expression tree** G as a graph (V, E) with vertex set $V = v_1, v_2, \ldots, v_n$, and edge set E. To each $v_i \in V$ we programmatically assign an atomic function, within the constraints defined by the BNF grammar.
$\lvert x \rvert$	We define the **cardinality** $\lvert x \rvert$ of a set, array or list x to be the number of elements in x.
$\overline{x}$	We define the **mean value** $\overline{x}$ of x to be the standard arithmetic mean $\frac{x_1+x_2+\cdots+x_n}{n}$.
$\min(x)$	We define the **minimum value** $\min[x]$ of x to be the *least* real value which x can assume, as predefined or by virtue of its character.
$\max[x]$	We define the **maximum value** $\max[x]$ of x to be the *greatest* real value which x can assume, as predefined or by virtue of its character.
λ	We define λ to be the **null character**.

Glossary

API *Application Programming Interface.* A set of routines, protocols and tools for the building of software applications. . . . 1

BNF *Backus-Naur Form.* A notation technique for context-free grammars; used herein to describe the syntax of binary expression trees. . . . 2–4, 8

GA *Genetic Algorithm.* A search heuristic, analogous in method to naturally-occurring genetic operations (*crossover*, *mutation* and *replication*). . . . 1, 2, 7, 8

GP *Genetic Programming.* The iterative transformation of a population of individuals (programs) into a new generation of individuals by the application of **GA**s. . . . 1–3, 6, 7

List of Algorithms

An Optimised Genetic Algorithm for Competitive Robocode Agents

Abstract

In this paper, we present an application of genetic programming to evolve an agent for competitive use in the simulated Robocode environment.

Robocode behavioural strategies are represented as binary trees of atomic expressions, respectively corresponding to observations and actions in the Robocode environment; such expression trees comprise a space of representations whereupon genetic programming is employed to conduct a search. We explore how orthodox genetic techniques (*crossover*, *mutation* and *replication*) can be conducted upon these trees to induce 'robots' that, over an indefinite number of generations, act according to learned behaviour.

If there is to be a competition, there must be some basis for resolving it. It is also clear that the ***competition should be experienced based.***

– John Henry Holland, *Hidden Order: How Adaptation Builds Complexity* [7], *p.* 53.

1 Introduction

1.1 The Robocode simulator

Robocode is a simulation-based game, developed by IBM Alphaworks [13], in which competitors design virtual tanks (*robots*) that fight to destruction in a closed arena; robots must navigate the environment, avoiding being shot by adversaries whilst simultaneously attempting to locate and shoot these adversaries. Robocode has a developed API in Java which, in conjunction with standard libraries, is used to develop the Robot class file; for the purposes of this project, we shall restrict any Java implementation extending classes other than Robot (such as, for instance, the AdvancedRobot class).

1.2 Purpose and scope of this document

Principally, we intend for this document to provide a cohesive overview of the GeneticBot system; thus, the majority of this document is dedicated to the direct discussion of the design of the genetic algorithm (GA), and, less exhaustively, the implementation thereof.

The scope of this document is limited to: a brief discussion of *existing research*, which shall outline previous contributions to the generation of Robocode agents *via* genetic programming (GP); a preliminary analysis of the Robocode environment and its relevant constituents; a high-level, language-agnostic sequential analysis of the approach, with clear respect to all necessary aspects of GP; and a low-level, language-specific examination of the subsequent implementation of the GA.

1.3 Existing work and justification of the approach

The earliest investigations into GAs as applied to the `Robocode` simulator were conducted by Eisenstein[1] [4] and Wyatt *et al.* [25]; further substantive research into the similar application of GAs has been carried out by Shichel *et al.* [18]. The application of GP to `Robocode` has historically proved successful; with a concise implementation of GP, Sipper [20, 19] scored first-place in the `Robocode` *HaikuBot* tournament[2].

Some attempts at evolving `Robocode` agents *via* 'non-traditional' means (*i.e.,* in deviation from orthodox GP as detailed in (**2.2**)) have been carried out. The application of *neural networks* in facilitating learning-based evolution is discussed in Czajkowski *et al.* [2]; Stanley [22] presents *Neuro-evolution of Augmenting Topologies* (NEAT), an *evolving-topology* GA '[. . .] that outperforms most fixed-topology networks in competitive learning tasks' ([14], *p.* 6). Liang *et al.* [14] discuss the efficacy of neural networks – with specific focus on a particular configuration of the aforementioned NEAT technology – in comparison to traditional GAs, concluding that the reason for their experimentally-observed failure of neurally-evolved robots '[. . .] could be that the actions performed by robots in `Robocode` are far too complex to learn with Neural Networks' ([14], *p.* 9).

Considering therefore the discrepancy between the complexity in implementation of neural networks – of both *fixed-* and *unfixed*-topology varieties –and the observed efficacy of neurally-evolved robots, we proceed to evolve `Robocode` agents through more traditional means, as detailed in (**2**).

Pre-programmed 'blocks' of behavioural strategies – 'activated' where appropriate in accordance with information obtained by the agent (see (**2.2**)) – have been utilised to success by Hong *et al.* [8], and similarly by Harper [5] in conjunction with a spatial co-evolution environment. While the approach has historically proved efficacious, it relies, whether used solely or in part, on *a priori* knowledge of optimal or near-optimal strategies for a considerably large range of scenarios within the `Robocode` environment. It is desirable, therefore, that similar results be obtained with minimal experimental analysis of the environment, which the traditional approach has been shown to provide.

The success of Harper's approach was facilitated also by the use of a BNF grammar, as translated into `Java`; this success, along with the comparable success of the unique approach detailed in Inja [9], motivates our own use of an expressive implemented grammar.

2 Analysis and Design

2.1 The atomic composition of `Robocode` robots

2.1.1 Actuators

The *active* behaviour of any particular iteration of the agent (any instance of the learned robot as generated by the GA, and any auxiliary files) is constituted by a number of methods to be directly called in the respective class file of the agent. We detail these active methods – whose arguments are precisely the expression trees (necessarily non-fixed among generations) whose structures are manipulated genetically – in (**3**).

Any agent, by virtue of these methods, may perform a number of actions. The robot body may be caused to rotate in its entirety, or to any programmatically-determined degree; alternatively, its gun-radar turret or the atomic radar component alone may be caused to similarly rotate, with proportional periods time allocated to performing each element[3]. Further, the robot may, at fixed

[1]Eisenstein contends that `Java` is not suitable for GP, noting in particular `Java`'s lack of an intrinsic tree-like structure that should facilitate the development of expression trees ('[. . .] genetic programming is more frequently used to evolve `Lisp`-like programs, which are weakly typed and are organized in a tree structure that is well-suited to crossover and mutation' ([4], *p.* 4)). Nonetheless, we circumvent this constraint by generating expressions in a procedural manner that effectively mimics the construction of binary trees.

[2]Information regarding the tournament can be found at `http://robocode.yajags.com/`.

[3]That is, the periodic rotation of the atomic radar component consumes markedly less time than the equivalent rotation of the entire robot body. However, we do not allow temporal factors to dictate or influence the composed expressions – and thus the active behaviour – which the appropriate methods subsume as arguments.

rates of distance and velocity, be caused to move forwards or backwards (if provided a negative argument) from its relative position; the robot is restricted from traversing the environment laterally, respective to its heading at any point.

In addition, the robot may, with a variable degree of *power* (where power correlates positively with damage inflicted, and negatively with the *energy* subsequently available to the agent), fire its turret wherever prompted.

2.1.2 Sensors

Any agent that should perform competently in a competitive `Robocode` environment must behave *reactively*, in conjunction with the aforementioned active behaviours.

The logic of any reactive behaviour is dictated by information obtained from the robot's lone *radar sensor.* The sensor, correctly utilised, thus provides the agent with all possible information that can be obtained about its adversaries; exhaustively, this information pertains to the agent's opponents' *position*, *heading*, *bearing*, *energy*, *velocity* and *distance of seperation* relative to the agent itself. We include the reactive methods whose returned values intuitively represent each of these data as legal expressions in the BNF grammar (see (**A**)).

2.2 Genetic programming

2.2.1 GP as a high-level abstraction

Proposed by John Holland in his seminal work *Adaption in Natural and Artificial Systems* [6], *genetic programming* (GP) can be characterised as a stochastic search algorithm which maintains a population of *individuals,* and simultaneously samples the *search space* $\mathcal{S}$. Following notation employed by Watanabe [24], we denote by $\Pi(t_0)$ a population of μ individuals $\psi_1(t_0), \ldots, \psi_\mu(t_0)$ for some initial *generation* t_0, where $\psi_i(t_0) \in \mathcal{S}$. Each individual $\psi(t_0)$ represents a *potential solution* to the concerned problem; each solution is evaluated to determine its measure of *fitness* (assigned by the *fitness function*), $\Phi(\psi_i(t_0))$. Programmatically-determined solutions are thus implemented as some complex data structure, denoted henceforth as ξ.

Subsequent to any necessary initialisation of the population, a series of *genetic operations – crossover, mutation* and *selection* – are applied to these structural representatives of the initial population, in order to obtain a new, resultant population. We can consider some crossover function ζ to constitute a higher-order transformation, $\zeta : (\xi \times \xi)^\mu \mapsto \xi^\lambda$, generating a population of *offspring* whose size is denoted locally by λ. Similarly, we may consider the mutation operation, whose corresponding function we denote by ζ', to constitute a unary transformation $\zeta' : \xi^\lambda \mapsto \xi^\lambda$ which 'mutates' the offspring population by applying an arbitrarily small change (that is, some trivial alteration in the structural representation of each individual). Considered as a function ω, the replication operation constitutes the transformation $\omega : (\psi^\lambda \cup \psi^{\mu+\lambda}) \mapsto \psi^\mu$, whereby a 'parent population' is determined for the subsequent generation $t_0 + 1$.[4]

We adapt the following pseudocode from Watanabe ([24], *p.* 6) in order to illustrate the high-level GP process:

[4]As discovered by Wyatt *et al.,* the 'elitist' approach to replication – replicating the individuals of absolute best-fit into the resultant generation – proves problematic with small populations of individuals ([25], *p.* 4). However, our approach to the problem at hand makes use of relatively large populations.

Algorithm 1: `HighLevelGA()`

```
variable $t_0 \leftarrow 0$                                   ▷ Initialise the generation counter

begin
    population $\Pi(t_0) \leftarrow \{\psi_1(t_0), \ldots, \psi_\mu(t_0)\}$
    evaluate $\Pi(t_0) : \{\Phi(\psi_1(t_0)), \ldots, \Phi(\psi_\mu(t_0))\}$
    while $t_0 \leq \max[t]$
        crossover $\Pi'(t_0) \leftarrow \zeta(\Pi(t_0))$
        mutate $\Pi''(t_0) \leftarrow \zeta'(\Pi'(t_0))$
        evaluate $\Pi''(t_0) : \{\Phi(\psi''_1(t_0)), \ldots, \Phi(\psi''_\mu(t_0))\}$
        replicate $\Pi(t_0 + 1) \leftarrow \omega(\Pi''(t_0) \cup Q)$, where $Q \in \{0, \Pi(t_0)\}$
        $t_0 \leftarrow t_0 + 1$
    end
end
```

It is expected that, after some indefinite number of generations ($\max[t]$), the procedure converges[5]; thus, the fittest individual for the final generation is expected to represent a near-optimum solution.[6]

2.2.2 A sequential overview of the approach

The complex data structures representing solutions to the problem at hand are, as is orthodox procedure in genetic programming, binary syntax trees, or *expression* trees. Thus, the fundamental objective of our approach is precisely the generation – and subsequent direct interpretation by the `Robocode` agent – of such trees, whose manipulable structure proves conducive to a process typified by repeated transformation.

Expression trees are comprised at each vertex of some atomic expression, where the class of the expression (as defined in the BNF grammar; see (**A**)) is determined by the character of the vertex; we allocate constants and variables to each *terminal* vertex (each vertex v of arity $a(v) = 0$), and arithmetic operations and (potentially multi-argumented) Java functions to each *function* vertex (of arity $a(v) > 0$), a category which subsumes by definition the root vertex. We illustrate this behaviour in **Figure 1**.[7]

The initial step of the approach, therefore, can be considered the development of a grammar, which we build constructively from its irreducible elements. The grammatical categories ⟨real⟩and ⟨bool⟩, which we define as follows, respectively represent necessarily atomic real and boolean values, and thus constitute possible expressions for allocation to terminal vertices of any generable expression tree:

$$\begin{aligned}
\langle\text{real}\rangle &\models x : x \in \mathbb{R} \\
\langle\text{bool}\rangle &\models \texttt{true} \mid \texttt{false} \mid \langle\text{bool}\rangle \texttt{ || } \langle\text{bool}\rangle \mid \langle\text{bool}\rangle \texttt{ \&\& } \langle\text{bool}\rangle \mid \texttt{!}\langle\text{bool}\rangle \\
\langle\text{cnst}\rangle &\models \texttt{0.001} \mid \texttt{Math.PI} \mid \texttt{Math.random()} \mid \texttt{Math.floor(Math.random()*}\langle\text{real}\rangle\texttt{)} \\
&\quad \mid r : r \in \mathfrak{R} \mid \varphi \mid \psi
\end{aligned}$$

[5]We observe that a larger population of candidates naturally results in greater variance across the fitness landscape, and thus a later point of convergence; we thus restrict our population size accordingly.

[6]In line with biological usage of the terms, we refer to such solutions – comprising a directly implementable, concrete instantiation of an abstract solution – as *phenotypes*, whose subsumed set of abstract properties we refer to as *genotypes*.

[7]In practice, fully-generated expression trees are considerably larger than depicted in **Figure 1**.

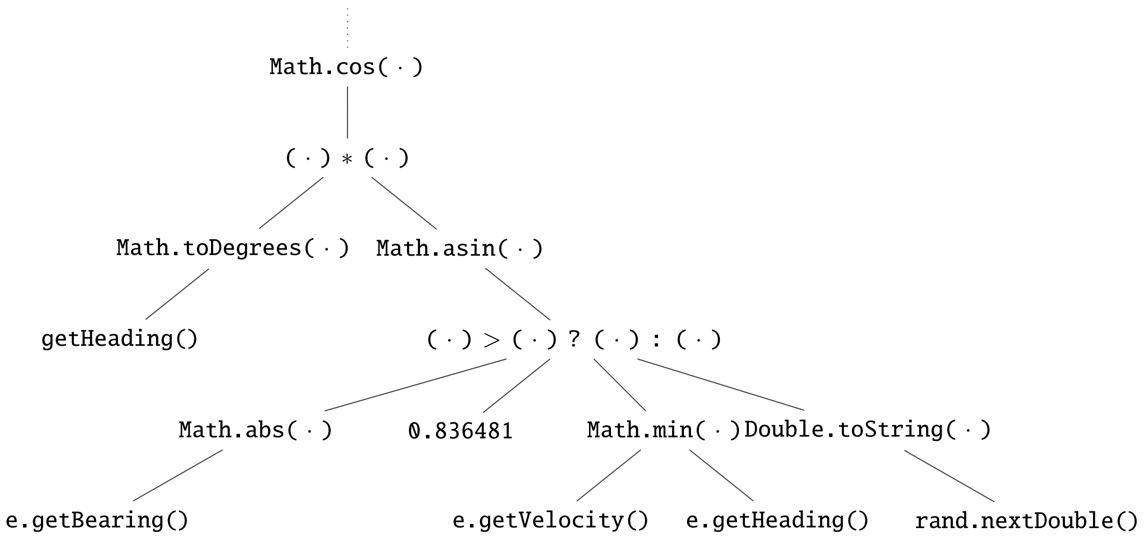

Figure 1: One of many possibly-generable expression trees.

Conversely, expressions of the category ⟨cnst⟩ may comprise a multi-argumented `Math.floor()` operation, and thus may constitute a function vertex. In our approach, we establish parameters for the assignment of valid atomic expressions to vertices; expression assignment is conducted proportionally, rather than pseudorandomly.

We construct trees in accordance with these parameters, ensuring iteratively that the scope of assigned expressions is restrained (and thus that the greater expression is unambiguous); structural representations of candidate solutions, determined with repeated comparisons between chromosomal fitness measures (see (**2.2.3**)), are thus prepared for the evolutionary process as detailed in (**2.2.1**).

Finally, subsequent to the necessary genetic operations, the fully-generated expressions are directly implemented, so as to be interpretable by the `Robocode` agent.

2.2.3 The *fitness* measure

The express purpose of the fitness function $\mathcal{F}$, in assigning a measure of fitness to each candidate solution and thus determining the trajectory of the evolutionary process, is the production of an agent of highest competence; that is, an agent of greatest *score*. It is essential, therefore, that the function should incorporate a translation of the appropriate data as obtained from the `Robocode` environment. Hence, we take this data to be comprised straightforwardly of all components that contribute towards the score, as demarcated by the Robocode documentation[8].

We denote by $S_{\psi(t)}$ the score of the agent at generation t, obtained in the manner previously detailed; by $S_{\overline{\psi(t)}}$, we denote the score of its opponent. The basis of the fitness measure, *pace* Hong *et al.* ([8], *p.* 637), is thus as follows:

$$\mathcal{F}:\ \psi(t) \mapsto \Phi(\psi(t)), \qquad \Phi(\psi(t)) = \frac{S_{\psi(t)}}{S_{\psi(t)} + S_{\overline{\psi(t)}}}.$$

Contra Hong *et al.*, however, we observe that, should any agent fail to obtain any score at all, the measure of fitness at present is not adequate.[9] As the probability of such an occurrence is

[8]The total score is the summation of seven statistics, detailed at `http://robowiki.net/wiki/Robocode/Scoring`.

[9]This inadequacy is similarly rectified in Inja [9], *p.* 11: the following fitness measure is introduced:

$$\Phi(\psi(t)) = 100 \cdot \big(1 + \frac{S_{\psi(t)} - S_{\overline{\psi(t)}}}{S_{\psi(t)} + S_{\overline{\psi(t)}}}\big).$$

nontrivial, we circumvent this limitation by introducing an arbitrarily small real-valued constant ϵ to the fractional measure:

$$\Phi(\psi(t)) = \frac{\epsilon + S_{\psi(t)}}{\epsilon + (S_{\psi(t)} + S_{\overline{\psi(t)}})}, \quad \epsilon \in \mathbb{R}^+.$$

Hence, any fitness valuations of $\psi(t)$ for which $\Phi(\psi(t)) \geq 0.5$ suggests a competency of the agent equal to or greater than its adversary.

2.2.4 The *selection* method

In our approach to strict fitness-proportional selection we adopt the *tournament selection* method, whereby a set of τ chromosomes (a subset $\pi(t_0) \subseteq \Pi(t_0)$) are abitrarily chosen and compared, with the fittest such chromosome populating the resultant set $\Pi'(t_0)$.

We adapt the following pseudocode from Blickle *et al.* ([1], *p.* 374) to illustrate our selection method:

```
Algorithm 2: TournamentSelection($\Pi(t_0), \tau$)
Input:  initial population of individuals $\Pi(t_0) = \{\psi_1(t_0), \dots, \psi_\mu(t_0)\}$
        constant $\tau \in \{1, 2, \dots, \mu\}$                ▷ The tournament size
for $i \leftarrow 1$ to $\mu$ do
    $\psi'_i(t_0) \leftarrow$ fittest individual from a random subset $\pi(t_0) \subseteq \Pi(t_0)$, $|\pi(t_0)| = \tau$
end
Output: resultant population of individuals $\Pi'(t_0) = \{\psi'_1(t_0), \dots, \psi'_\mu(t_0)\}$
```

The tournament selection approach is motivated by strong support of its efficacy and optimality in the literature, particularly when compared with other selection methods. In *Genetic Algorithms,* Reeves plainly determines tournament selection to be an optimal approach to GP problems when considered alongside the 'roulette-wheel' selection approach[10] ([17], *p.* 133); Wyatt *et al.* adopt tournament selection in their genetic evolution of `Robocode` agents, citing a desired avoidance of '[...] the problems associated with the roulette wheel method' ([25], *p.* 4).

The tournament *size parameter* τ, in positively correlating with a loss in genetic diversity[11], must be chosen with some degree of precaution. Historically, successful attempts at genetically evolving `Robocode` agents have been conducted with size parameters as low as $\tau = 2$ (*cf.* Wyatt *et al.* [25], *p.* 4) and as high as $\tau = 5$ (*cf.* [19], *p.* 93); thus, we adopt the discrete middle ground and conduct our selection procedure with a size parameter of $\tau = 4$.

2.2.5 Probabilities for genetic transformations

The optimisability of probabilities for genetic transformations is well-studied[12]. We consider rates for primary genetic operations – crossover, mutation and replication – that are likely to produce, subsequent to their application, competitive `Robocode` agents.

Crossover In accordance with the rate suggested by Koza [11] – and implemented successfully by Eisenstein [4] – we adopt a crossover probability of $p_C = 0.87$, squarely within the 'typical' range $[0.75, 0.95]$ (*cf.* [25], *p.* 4). *Pace* Koza *et al.* [12] (*pp.* 138–139), we divide

[10]The 'roulette-wheel' approach uses a probability distribution for selection, in which the selection probability of a given chromosome is proportional to its fitness (*cf.* Reeves [17], *pp.* 120–121).

[11]As detailed in Sokolov *et al.* [21], *pp.* 1132–1133. Motoki computes the loss in genetic diversity $D_{loss}(\mu, \tau)$ as follows ([15], *p.* 401):

$$D_{loss}(\mu, \tau) = \frac{1}{\mu} \cdot \sum_{k=1}^{\mu} (1 - \frac{k^\tau - (k-1)^\tau}{\mu^\tau})^\mu.$$

[12]An overview of the literature is presented in Patil *et al.* [16].

the holistic rate of the crossover operation into three distinct probabilities; $p_{CF} = 0.60$, $p_{CR} = 0.30$ and $p_{CT} = 0.10$, corresponding to the potential crossover point of the *function* vertex[13], the *root* vertex and the *terminal* vertex respectively. The many-point crossover approach – in contrast to single-point crossover – is motivated by a thorough analysis by Eshelman, citing in particular the associated minimisation of positional and distributional bias exploitation within the GA. A similar sentiment, favouring many-point and other alternatives over single-point crossover, is offered in Reeves ([17], *p.* 133).

Mutation A $p_M = 0.01$ probability of mutation is typical in solving GP-related problems[14], and is outrightly suggested by Koza *et al.* ([12], *p.* 137). In an anlysis of rate-optimality conducted by DeJong, it is determined that a rate of mutation $p_M > 0.05$ is in general harmful for the optimal performance of GAs[15] ([3], *p.* 67–68); as a mediary solution, and thus in accordance with the rate suggested by Koza [11], we adopt a $p_M = 0.03$ probability of mutation.

Replication The remaining probability is thus allocated to the process of replication, and we adopt the rate $p_R = 0.10$.

2.2.6 Characteristics of expression trees

Preliminary constraints on the generation of expression trees are minimal; only the minimum and maximum depths need to be determined. Following Wyatt *et al.*, on the observation that a maximum depth of 10 provides '[. . .] enough diversity at first to produce competitive tanks' (Wyatt *et al.* [25], *p.* 4), we adopt the same maximum depth parameter, and similarly a minimum depth parameter of 3.

3 Implementation

For the main class file, interpreted by the agent, we adopt the method detailed in Sipper [19], *p.* 90. Generated expression trees $\xi_i(t_0)$ for the initial generation, as illustrated in the below excerpt, compose the arguments of the agent's primary event methods; thus, they comprise a series of actions determining the behaviour detailed in (**2.1**).

Listing 1: `GeneticBot-0.0`

```
public class GeneticBot-0.0 extends Robot {

  public void run() {
    while(true) {
      turnGunRight(∞);
    }

  public void onScannedRobot(ScannedRobotEvent e) {
    setAhead(ξ1(t0));
    setTurnRight(ξ2(t0));
    setTurnGunRight(ξ3(t0)⟩);
    setTurnRadarRight(ξ4(t0));
    setFire(ξ5(t0));
  }

}
```

[13]The variable position of the function vertex is illustrated in **Figure 1**.

[14]Urbano *et al.*, for instance, use a 'standard' mutation of rate of 0.01 to success ([23], *p.* 922).

[15]Further, it is concluded that, provided optimal control parameters for a GA with single-chromosome representation, a too-high mutation rate renders the efficacy of the stochastic GP search comparable to a random search, irrespective of auxiliary parameter settings (as illustrated by graphical data; [3], *p.* 71).

Notably, such expression trees may not necessarily remain discrete amongst action methods; the evolutionary process, as in accordance with predefined parametric conditions (see (**2.2.5**)), may permit genes to cross between trees, and the replication of subtrees may occur between arguments.

A number of class files have been developed in order to ultimately produce iterations of the learned agent across generations; we illustrate the relationship between these classes in **Figure 2**. The low-level `Genotype` class is responsible for the characterisation of expression trees (*i.e.,* the candidates $\psi(t_i)$), including the programmatic translation of legal expressions defined by the BNF grammar, the establishing of parameters governing the assignment of these expressions, and the fundamental, largely recursive initialisation methods upon which the trajectory of the evolutionary process is contingent (the `compose()` and `expand()` methods, for instance, govern the depth-first traversal-based composition of `String` expressions and the recursive expansion of expression trees, respectively). The translation of the grammar into `Java` is accomplished by virtue of multi-dimensional `String` arrays, with higher-dimension arrays subsuming grammatical categories of lower complexity.

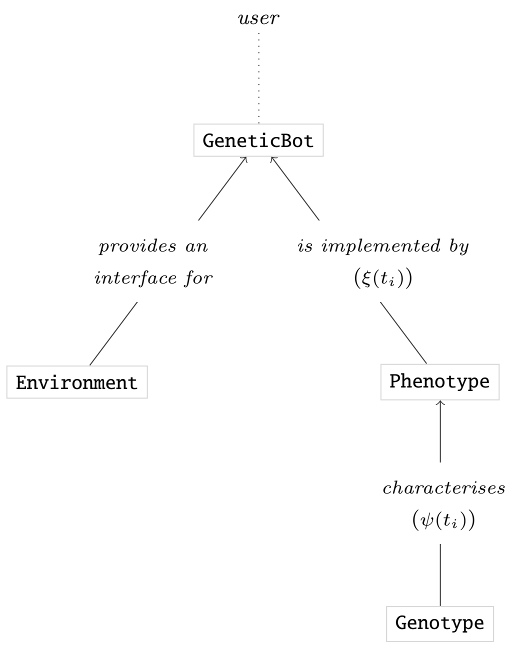
user
GeneticBot
provides an interface for
is implemented by ($\xi(t_i)$)
Environment
Phenotype
characterises ($\psi(t_i)$)
Genotype

Figure 2: Relations between classes

The `Phenotype` class, in representing the production of the pseudo-structures $\xi(t_i)$, utilises (as an instantiated object) an array of `Genotype`s that function as root vertices of the expression trees[16]. `Phenotype` is responsible for the generation of the source code for each iteration of the `Robocode` agent, directly interpreted by the agent within the environment; this is accomplished straightforwardly by isolating the generic frame common to extensions of the `Robot` class, including the structures $\xi(t_i)$ where appropriate.

The `GeneticBot` class assumes the essential form of the high-level GA observed in (**2.2.1**), populating an array of `Phenotype`s representing the population of individuals. Further, `GeneticBot` handles the selection of genetic operations, in accordance with the rates determined and justified in (**2.2.5**).

The auxiliary `Environment` class acts as an interface for `Robocode` itself, initialising an engine and executing rounds within the `Robocode` environment, and hence causing the generation of agents for max[t] generations. The fitness measure $\Phi(\psi(t_i))$, in accepting as a variable the obtained scores of both the agent and its adversary, is herein included and calculated.

[16]At present, we limit the number of chromosomes to 5, all implemented solely within the `onScannedRobot` event.

References

[1] Blickle, T., and Thiele, L. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation 4*, 4 (1996), 361–394.

[2] Czajkowski, A., and Patan, K. Real-time learning of neural networks and its application to the prediction of opponent movement in the Robocode environment. In *Proceedings of the XI International PhD Workshop*.

[3] DeJong, K. A. Analysis of the Behavior of a Class of Genetic Adaptive Systems.

[4] Eisenstein, J. Evolving Robocode Tank Fighters. Tech. Rep. 2003-023, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA, 2003.

[5] Harper, R. Co-Evolving Robocode Tanks. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2011), SIGEVO, ACM, pp. 1443–1450.

[6] Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, Ann Arbor, MI, USA, 1975.

[7] Holland, J. H. *Hidden Order: How Adaptation Builds Complexity.* Helix Books. Addison-Wesley Publishing Company, Boston, MA, USA, 1995.

[8] Hong, J.-H., and Cho, S.-B. Evolution of Emergent Behaviors for Shooting Game Characters in Robocode. In *Proceedings of the Congress on Evolutionary Computation.*

[9] Inja, M. T. Genetic Programming and Robocode. Bachelor's thesis, Faculty of Science, University of Amsterdam, Amsterdam, Netherlands, 2012.

[10] Koza, J. R. Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Tech. Rep. STAN-CS-90-1314, Department of Computer Science, Stanford University, Stanford, CA, USA, 1990.

[11] Koza, J. R. *Genetic Programming: on the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[12] Koza, J. R., and Poli, R. *Genetic Programming.* Springer, New York, NY, USA, 2005.

[13] Li, S. Rock 'em, sock 'em Robocode!, 2002.

[14] Liang, R., and Zhao, P. Applying and Comparing Evolutionary Algorithms for Robot Tanks. Tech. rep., Department of Computer Science, Swarthmore College, Swarthmore, PA, USA, 2014.

[15] Motoki, T. Calculating the expected loss of diversity of selection schemes. *Evolutionary Computation 10*, 4 (2002), 397–422.

[16] Patil, V. P., and Pawar, D. D. The optimal crossover or mutation rates in genetic algorithm: A review. *International Journal of Applied Engineering and Technology 5*, 3 (2014), 38–41.

[17] Reeves, C. R. *Genetic Algorithms*, second ed., vol. 146 of *International Series in Operations Research & Management Science.* Springer, 2010, pp. 109–140.

[18] Shichel, Y., Ziserman, E., and Sipper, M. *GP-Robocode: Using Genetic Programming to Evolve Robocode Players*, vol. 3447. Springer-Verlag, Heidelberg, Germany, 2005, pp. 143–154.

[19] Sipper, M. *Evolved to Win.* Lulu, Raleigh, NC, USA, 2011.

[20] Sipper, M. *Let the Games Evolve!* Genetic and Evolutionary Computation. Springer-Verlag, New York, NY, USA, 2011, pp. 17–36.

[21] Sokolov, A., and Whitley, D. Unbiased tournament selection. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation* (2005), ACM, pp. 1131–1138.

[22] Stanley, K. O., and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary Computation 10*, 2 (2002), 99–127.

[23] Urbano, P., and Georgiou, L. Improving Grammatical Evolution in Santa Fe Trail using Novelty Search. *Advances in Artificial Life, ECAL 12* (2013), 917–924.

[24] Watanabe, K., and Hashem, M. M. A. *Evolutionary Computations: New Algorithms and their Applications to Evolutionary Robots*, vol. 147 of *Studies in Fuzziness and Soft Computing.* Springer-Verlag, Heidelberg, Germany, 2004.

[25] Wyatt, D., and Klein, D. Genetic Programming for `Robocode` Strategy. Tech. rep., Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, 2003.

Appendix A Complete BNF Grammar

The vertices of constructed expression trees represent atomic expressions (constants, bases of conditionals, *etc.*) that conform to the categories defined programmatically (see **Algorithm 5**). Herein we include the complete grammar, providing a basis for the demarcation of those categories.

We define, *pace* Koza [10, 11], the ephemeral random constants $\mathfrak{R} \in V$:

> Whenever the ephemeral atom “$\mathfrak{R}$” is chosen for any point of the tree during the generation of the initial random population, a random number in a specified range is generated and attached to the tree at that point. In a real-valued problem [. . .], the random constants [are] floating point numbers between `-1.0` and `+1.0`. In a problem involving integers (*e.g.* induction of a sequence of integers), random integers over a specified range are generated for the ephemeral “$\mathfrak{R}$” atoms. In a Boolean problem, the ephemeral “$\mathfrak{R}$” atoms are replaced by either `T` (True) or `NIL`’ (Koza [10], *p.* 40).

$$
\begin{array}{rcl}
\langle\text{real}\rangle & \models & x : x \in \mathbb{R} \\
\langle\text{bool}\rangle & \models & \textbf{true} \mid \textbf{false} \mid \langle\text{bool}\rangle\ \texttt{||}\ \langle\text{bool}\rangle \mid \langle\text{bool}\rangle\ \texttt{\&\&}\ \langle\text{bool}\rangle \mid \texttt{!}\langle\text{bool}\rangle \\
\langle\text{cnst}\rangle & \models & \texttt{0.001} \mid \texttt{Math.PI} \mid \texttt{Math.random()} \mid \texttt{Math.floor(Math.random()*}\langle\text{real}\rangle\texttt{)} \\
 & & \mid r : r \in \mathfrak{R} \mid \varphi \mid \psi \\
\langle\text{univ}\rangle & \models & \texttt{getEnergy()} \mid \texttt{getVelocity()} \\
 & & \mid \texttt{getHeight()} \mid \texttt{getWidth()} \\
 & & \mid \texttt{getHeading()} \mid \texttt{getBearing()} \\
 & & \mid \texttt{getX()} \mid \texttt{getY()} \\
\langle\text{evnt}\rangle & \models & \texttt{e.getEnergy()} \mid \texttt{e.getVelocity()} \\
 & & \mid \texttt{e.getHeading()} \mid \texttt{e.getBearing()} \\
 & & \mid \texttt{e.getDistance()} \\
\langle\text{bnry}\rangle & \models & \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \\
\langle\text{oper}\rangle & \models & \langle\text{real}\rangle \mid \langle\text{cnst}\rangle \mid \langle\text{univ}\rangle \mid \langle\text{evnt}\rangle \mid \langle\text{func}\rangle \\
\langle\text{func}\rangle & \models & \texttt{Math.sin(}\langle\text{oper}\rangle\texttt{)} \mid \texttt{Math.asin(}\langle\text{oper}\rangle\texttt{)} \\
 & & \mid \texttt{Math.cos(}\langle\text{oper}\rangle\texttt{)} \mid \texttt{Math.acos(}\langle\text{oper}\rangle\texttt{)} \\
 & & \mid \texttt{Math.toDegrees(}\langle\text{oper}\rangle\texttt{)} \mid \texttt{Math.toRadians(}\langle\text{oper}\rangle\texttt{)} \\
 & & \mid \texttt{Math.min(}\langle\text{oper}\rangle\texttt{)} \mid \texttt{Math.max(}\langle\text{oper}\rangle\texttt{)} \mid \texttt{Math.abs(}\langle\text{oper}\rangle\texttt{)} \\
 & & \mid \langle\text{func}\rangle\langle\text{bnry}\rangle\langle\text{oper}\rangle \\
\langle\text{asgn}\rangle & \models & \varphi\ \texttt{=}\ \langle\text{stmt}\rangle \mid \varphi\ \texttt{!=}\ \langle\text{stmt}\rangle \\
\langle\text{cond}\rangle & \models & \textbf{if}\ \langle\text{expr}\rangle\ \textbf{then}\ \langle\text{stmt}|\text{asgn}\rangle\ \textbf{else}\ \langle\text{stmt}|\text{asgn}\rangle \\
 & & \mid \textbf{while}\ \langle\text{expr}\rangle\langle\text{stmt}|\text{asgn}\rangle \\
\langle\text{stmt}\rangle & \models & \langle\text{univ}\rangle \mid \langle\text{evnt}\rangle \mid \langle\text{cond}\rangle \mid \langle\text{stmt}\rangle\langle\text{bnry}\rangle\langle\text{stmt}\rangle \\
 & & \textbf{return}\ \langle\text{real}|\text{bool}|\text{cnst}|\text{func}\rangle \mid \textbf{return}\ \lambda \\
\langle\text{expr}\rangle & \models & \mid \langle\text{real}\rangle \mid \langle\text{bool}\rangle \mid \langle\text{func}\rangle \mid \langle\text{asgn}\rangle \mid \langle\text{stmt}\rangle
\end{array}
$$

Appendix B Complete Pseudocode Listings

We list herein pseudoclasses for the generation of a fully-functional, genetic `Robocode` agent.

B.1 The `GeneticBot` class

The high-level interface, by virtue of which the genetic components and operations are initialised and the executable agent $\Xi^{|\Pi(t_i)|}$ is generated.

Algorithm 3: `GeneticBot`$(\mathbf{O})$

```
Input: 1-dimensional array $\mathbf{O} = \{\Xi_1^{\mathbf{O}}, \Xi_2^{\mathbf{O}}, \dots, \Xi_n^{\mathbf{O}}\}$ of opposing pre-programmed agents
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
variables $t_0 \leftarrow t \leftarrow 0,\ t_0,\ t \in \mathbb{N}$    ▷ Generational counters, where $\psi(t_0/t) \in \mathcal{S}$
1-dim. array $\mathbf{F} \leftarrow \varnothing,\ |\mathbf{F}| = |\Pi(t_0)| = \mu$    ▷ Chromosomal fitnesses: $\mathbf{F}[i] = \Phi(\psi(t_i))$
             $\overline{\mathbf{F}} \leftarrow \varnothing,\ |\overline{\mathbf{F}}| = \max[t_0]$    ▷ Mean fitnesses: $\overline{\mathbf{F}}[i] = \overline{\Phi(\psi(t_i))}$
             $\overline{\mathbf{N}} \leftarrow \varnothing,\ |\overline{\mathbf{N}}| = \max[t_0]$    ▷ Mean no. of vertices: $\overline{\mathbf{N}}[i] = \overline{\sum v :\ \forall v \in V_{\Pi(t_i)}}$
             $\mathbf{M} \leftarrow \mathbf{M}' \leftarrow$ Phenotype[$\mu$]    ▷ Populations: $\mathbf{M}[i] = \Pi(t_i)$
             $\mathfrak{P} \leftarrow$ Phenotype[$\max[t_0]$]    ▷ The $\underline{\textit{phenome}\text{ set of phenotypes}}$[a]: $\mathfrak{P}[i] = \xi(t_i)$
variables $F \leftarrow \sum_{\alpha \leftarrow 0}^{\mu} \Phi(\psi_\alpha(t_0)),\ \overline{F} \leftarrow \overline{\sum_{\alpha \leftarrow 0}^{\mu} \Phi(\psi_\alpha(t_0))},\ \overline{N} \leftarrow \sum_{\alpha \leftarrow 0}^{|V_{\Pi(t_0)}|} v_\alpha :\ \forall v \in V_{\Pi(t_0)}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
individual $\xi(t_0) \leftarrow$ Phenotype($-1$), $\Phi(\xi(t_0)) \leftarrow 0$    ▷ The optimal solution (phenotype)
for $i \leftarrow 0$ to $\mu$ do
    $\mathbf{M}[i] \leftarrow$ Phenotype(0)    ▷ Initialise the population
    for $j \leftarrow 0$ to PHENOTYPE.$|\mathfrak{p}|$ do
        PHENOTYPE.$\mathfrak{g}[j] \leftarrow$ Genotype(0)
        PHENOTYPE.$\mathfrak{g}[j]$.setArity(0)
        PHENOTYPE.$\mathfrak{g}[j]$.assignExpr(0, 0)
    end
end
foreach $\psi(t_i) \in \mathbf{M}[i]$ do
    for $j \leftarrow 0$ to PHENOTYPE.$|\mathfrak{p}|$ do
        PHENOTYPE.$\mathfrak{p}[j] \leftarrow$ PHENOTYPE.$\mathfrak{g}[j]$.compose()
        Implement data structures characterised by PHENOTYPE.$\mathfrak{p}[\alpha],\ \forall \alpha \in \{0, \dots, |\mathfrak{p}|\}$
    end
end
while $t_0 < \max[t_0]$
    Populate $\mathbf{F}$ with fitnesses for $R$ rounds, where $R \in \mathbb{N}$ is user-determined
    variable $\phi(t_0) \leftarrow 0$    ▷ "Competitor" chromosome; potential solution (potential phenotype)
    for $i \leftarrow 0$ to $\mu$ do
        $\Phi(\mathbf{M}[\phi(t_0)]) \leftarrow \mathbf{F}[i],\ F \leftarrow F + \Phi(\mathbf{M}[\phi(t_0)])$
        if $\Phi(\mathbf{M}[i]) > \Phi(\mathbf{M}[\phi(t_0)])$ then $\phi(t_0) \leftarrow i$    ▷ Iterative comparison to identify phenotypes
        $\overline{N} \leftarrow \overline{N} + \sum_{\alpha \leftarrow 0}^{|V_{\Pi(t_0)}|} v_\alpha : \forall v_\alpha \in V_{\Pi(t_0)}$
    end
    ...
```

[a] Directly comparable to PHENOTYPE.$\mathfrak{p}$.

```
...
    $\overline{F} \leftarrow {}^{F}/_{\mu},\ \overline{\mathbf{F}}[t_0] \leftarrow \overline{F}$
    $\overline{N} \leftarrow {}^{\overline{N}}/_{\mu},\ \overline{\mathbf{N}}[t_0] \leftarrow \overline{N}$
    $\mathfrak{P}[t_0] \leftarrow \mathbf{M}[\phi(t_0)]$    ▷ Store the most competitive chromosome as the generational phenotype
    if $\Phi(\mathbf{M}[\phi(t_0)]) > \Phi(\xi(t_0))$ then $\xi(t_0) \leftarrow \mathbf{M}[\phi(t_0)]$
    $t_0 \leftarrow t_0 + 1$
    breed()
    $\mathbf{M} \leftarrow \mathbf{M}',\ \mathbf{M}' \leftarrow$ Phenotype[$\mu$]
    foreach $\psi(t_i) \in \mathbf{M}[i]$ do
        for $j \leftarrow 0$ to PHENOTYPE.$|\mathfrak{p}|$ do
            PHENOTYPE.$\mathfrak{p}[j] \leftarrow$ PHENOTYPE.$\mathfrak{g}[j]$.compose()
            Implement data structures characterised by PHENOTYPE.$\mathfrak{p}[\alpha],\ \forall\alpha \in \{0,\dots,|\mathfrak{p}|\}$
        end
    end
end
Subroutine breed()
    exogeneous parameter $p_c \leftarrow x :\ x \in [0.6, 1.0]$, where $\zeta_{p_c} :\ (\xi \times \xi)^{\mu} \mapsto \xi^{\lambda},\ \lambda = |\Pi'(t_0)|$
                         $p_m \leftarrow x :\ x \leq 0.10$,   where $\zeta'_{p_m} :\ \xi^{\lambda} \mapsto \xi^{\lambda}$
    $\mathbf{M}'[0] \leftarrow \mathfrak{P}[t_0 - 1]$.replicate($t_0$, 0)
    $\mathbf{M}'[1] \leftarrow \xi(t_0)$.replicate($t_0$, 1)
    variable $i \leftarrow 2$
    while $i < \mu$
        Let $\rho$ be a pseudorandom variable following the uniform distribution over $[0.0, 1.0]$[a]
        if $(\rho \leftarrow \rho - p_c) \leq 0$ then
            variables $\tau \leftarrow \tau' \leftarrow$ tournament()
            $\mathbf{M}'[i] \leftarrow \mathbf{M}[\tau]$.crossover($\mathbf{M}[\tau']$, $t_0$, $i$)    ▷ Crossover $\zeta_{p_c} :\ (\tau \times \tau') \mapsto i$
        else if $(\rho \leftarrow \rho - p_m) \leq 0$ then
            $\mathbf{M}'[i] \leftarrow \mathbf{M}$[tournament()].mutate($t_0$, $i$)    ▷ Mutation $\zeta'_{p_m} :\ i \mapsto i$
        else
            $\mathbf{M}'[i] \leftarrow \mathbf{M}$[tournament()].replicate($t_0$, $i$)    ▷ Replication $\omega :\ (i^{\lambda} \cup i^{\mu+\lambda}) \mapsto i^{\mu}$
        end
        $i \leftarrow i + 1$
    end
Subroutine tournament()
    Let $\varsigma$ be a pseudorandom variable following the uniform distribution over $[0, \mu]$
    1-dim. array $\mathbf{T} \leftarrow \varnothing,\ |\mathbf{T}| = x :\ x \in \mathbb{N}$    ▷ Arbitrary chromosomes $\psi(t_0) \in \Pi(t_0)$
    variable $f \leftarrow \mathbf{T}[0]$    ▷ The fittest randomly selected phenotype
    for $i \leftarrow 0$ to $|\mathbf{T}|$ do
        $\mathbf{T}[i] \leftarrow \varsigma$
        if $i \neq 0$ then
            if $\Phi(\mathbf{M}[\mathbf{T}]) > \Phi(\mathbf{M}[f])$ then $f \leftarrow \mathbf{T}[i]$    ▷ Compare fitnesses to identify phenotypes
        end
    end
    return $f$
```

[a]Conventions for pseudorandom variables remains consistent throughout; let ρ follow a non-discrete distribution, and ς a discrete distribution.

Output: Implemented agent $\Xi^{|\Pi(t_0)|}$ subsuming the phenotypal solutions $\xi(t_0) \in \mathfrak{P}$, with mean fitness $\overline{\mathbf{F}}$ and $\sum_{\alpha \leftarrow 0}^{|V_{\mathfrak{P}[0]}|} v_\alpha : \forall v \in V_{\mathfrak{P}[0]}$ vertices (mean *no.* of vertices $\overline{\mathbf{N}}$)

B.2 The Phenotype class

The Phenotype pseudoclass operates upon expression trees – defined by the Genotype pseudoclass – which determine the behaviour of the Robocode agent.

Algorithm 4: Phenotype(t_0)

```
Input: variable $t_0 \in \mathbb{N}$                    ▷ The generational counter, pace Algorithm 3
-------------------------------------------------------------------------------
1-dim. array $\mathfrak{p} \leftarrow \varnothing$                    ▷ Phenotypes, i.e., expression trees
             $\mathfrak{g} \leftarrow \varnothing$     ▷ Genotypes: initial trees, upon which genetic operations are conducted
exogeneous parameter $p_{CJ} \leftarrow x : x \leq 0.05$      ▷ Crossover $\zeta_{p_{CJ}}$, crossover is non-automorphic[a]
                     $p_{CT} \leftarrow x : x \leq 0.10$      ▷ Crossover $\zeta_{p_{CT}}$, crossover point is a terminal
                     $p_{CR} \leftarrow x : x \in [0.2, 0.3]$ ▷ Crossover $\zeta_{p_{CR}}$, crossover point is the root vertex
                     $p_{MT} \leftarrow x : x \leq 0.15$      ▷ Mutation $\zeta'_{p_{MT}}$, mutated vertex is terminal
                     $p_{MR} \leftarrow x : x \leq 0.05$      ▷ Mutation $\zeta'_{p_{MR}}$, mutated vertex is the root vertex
variable $N \leftarrow 0$   ▷ No. of nodes for some individual $\psi(t_0) \in \Pi(t_0)$: $N \equiv \sum_{\alpha \leftarrow 0}^{|V_{\psi(t_0)}|} v_\alpha : \forall v \in V_{\psi(t_0)}$
-------------------------------------------------------------------------------
Subroutine crossover($\xi(t_0)$, $t_0$)
    Let $\rho$, $\rho'$ (init[$\rho$] $\neq$ init[$\rho'$]) be pseudorandom variables following the distribution over $[0.0, 1.0]$
        $\varsigma$, $\varsigma'$ (init[$\sigma$] $\neq$ init[$\sigma'$]) be pseudorandom variables following the distribution over $[0, |\mathfrak{p}|]$
    variables $\varphi \leftarrow \varsigma$, $\varphi' \leftarrow \varsigma'$
    individual $\xi'(t_0) \leftarrow$ Phenotype($t_0$)
    for $i \leftarrow 0$ to $|\mathfrak{p}|$ do $\xi'(t_0).\mathfrak{g}[i] \leftarrow \mathfrak{g}[i]$.clone()      ▷ Perform cloning operation
    if $\rho < p_{CR}$ then
        if $\rho < p_{CJ}$ then
            $\xi'(t_0).\mathfrak{g}[\varphi]$.replace($\xi(t_0).\mathfrak{g}[\varphi']$)      ▷ Perform non-automorphic replacement operation
        else
            $\xi'(t_0).\mathfrak{g}[\varphi]$.replace($\xi(t_0).\mathfrak{g}[\varphi]$)       ▷ Index-automorphic replacement
        end
    else
        boolean crosst$_1$ $\iff$ ($\rho < p_{CT}$)
                crosst$_2$ $\iff$ ($\rho' < p_{CT}$)
        $\xi'(t_0).\mathfrak{g}[\varphi]$.insert($\xi(t_0).\mathfrak{g}[\varphi]$.getSubTree(crosst$_1$))
        $\xi'(t_0).\mathfrak{g}[\varphi']$.insert($\xi(t_0).\mathfrak{g}[\varphi']$.getSubTree(crosst$_2$))
    end
  ...
```

[a]That is, conducted on non-equal chromosomal indices: $i \neq j$ for phenotypal indices $\mathfrak{p}_1[i], \mathfrak{p}_2[j]$, where $\mathfrak{p}_1, \mathfrak{p}_1$ are subsumed by the agent Ξ.

```
  ...
    $\xi'(t_0)$.setDepths(), $\xi'(t_0)$.countNodes()
    return $\xi'(t_0)$
Subroutine mutate($t_0$)
    Let $\rho$ be a pseudorandom variable following the uniform distribution over $[0.0, 1.0]$
        $\varsigma$ be a pseudorandom variable following the uniform distribution over $[0, |\mathfrak{p}|]$
    individual $\xi'(t_0) \leftarrow$ Phenotype($t_0$)
    if $\rho < p_{MR}$ then
        $\xi'(t_0).\mathfrak{g}[\varsigma] \leftarrow$ Genotype(0)
        $\xi'(t_0).\mathfrak{g}[\varsigma]$.setArity(0) ▷ Commence expansion of the expression tree: set arity of vertices
        $\xi'(t_0).\mathfrak{g}[\varsigma]$.assignExpr(0, 0)                        ▷ Assign expressions to vertices
    else if $\rho < p_{MT}$ then
        $\xi'(t_0).\mathfrak{g}[\varsigma]$.mutateTerm()                  ▷ Perform mutation on a terminal vertex
    else
        $\xi'(t_0).\mathfrak{g}[\varsigma]$.mutateFunc()                ▷ Perform mutation on a functional vertex
    end
    $\xi'(t_0)$.setDepths(), $\xi'(t_0)$.countNodes()
    return $\xi'(t_0)$
Subroutine replicate($t_0$)
    individual $\xi'(t_0) \leftarrow$ Phenotype($t_0$)
    for $i \leftarrow 0$ to $|\mathfrak{p}|$ do
        $\xi'(t_0).\mathfrak{g}[i] \leftarrow$ Genotype(0)
        $\xi'(t_0).\mathfrak{g}[i]$.replace($\mathfrak{g}[i]$)
    end
    $\xi'(t_0)$.setDepths(), $\sum_{\alpha \leftarrow 0}^{|V_{\xi'(t_0)}|} v_\alpha : \forall v \in V_{\xi'(t_0)} \leftarrow N$
    return $\xi'(t_0)$
Subroutine countNodes()
    $N \leftarrow 0$
    for $i \leftarrow 0$ to $|\mathfrak{g}|$ do $N \leftarrow N + \mathfrak{g}[i]$.countNodes()
    return $N$
Subroutine setDepths()
    foreach $\psi(t_0) \in \mathfrak{g}$ do $\psi(t_0)$.setDepths(0)
```

Output: *Genetic operations (crossover, mutation, replication):* Unimplemented data structure $\xi'(t_0)$ for the phenotypal solution of generation t_0
countNodes() *subroutine:* Number of nodes N for individual $\mathfrak{g}[i]$

B.3 The `Genotype` class

The `Genotype` pseudoclass governs the structure and behaviour of the Koza-type expression trees, excepting the performance of any genetic operations for which the `Phenotype` pseudoclass is responsible.

Note: *We define the* 0-generation parent *of a vertex v to be v itself.*

```
Algorithm 5: Genotype(d(v_0)/(d(v_0), a(v_0), terminal(v_0)))
Input: vertex depth $d(v_0) = |\{w : w \in V_{\Pi(t_i)} \wedge \text{child}(w) = (k\text{-generation parent}(v_0),\ k \in \mathbb{N}^0)\}|$
                arity $a(v_0) = |\{w : w \in V_{\Pi(t_i)} \wedge v_0 = (1\text{-generation parent}(w))\}|$
       boolean terminal($v_0$) $\iff (\neg\exists w : w \in V_{\Pi(t_i)} \wedge v_0 = (k\text{-generation parent}(w),\ k \in \mathbb{N}^+))$
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1-dim. array $\Sigma \leftarrow \varnothing$    ▷ Expressions: a well-formed expression is defined by the form $\langle\text{expr}\rangle$[a]
             $\mathbf{C} \leftarrow \varnothing$    ▷ Child vertices: $\{v : \exists w(k\text{-generation parent}(v) = w),\ k \in \mathbb{N}^+\} \to \mathbf{C}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$d(v_0) \leftarrow 0$, $a(v_0) \leftarrow -1$, terminal($v_0$) $\leftarrow$ true
Generate the 3-dim. array[b] $\langle\mathbf{E}\rangle = \{\varepsilon_{Term}, \varepsilon_{Func}, \varepsilon_{Bnry}, \varepsilon_{Cond}\}$ whose elements are n-dim. arrays
  ($n < 3$) representing the appropriate BNF-valid arguments[c]
Subroutine compose()
    string $\sigma \leftarrow \Sigma[0]$
    if $a(v_0) = -1$ then return $\lambda$
    for $i \leftarrow 0$ to $a(v_0)$ do $\sigma \leftarrow (\sigma + \mathbf{C}[i]$.compose() $+ \Sigma[i+1])$    ▷ Recursive function call
    sanitise $\sigma$
    return $\sigma$
Subroutine countNodes()
    variable $N \leftarrow 1$
    for $i \leftarrow 0$ to $a(v_0)$ do $N \leftarrow N + \mathbf{C}[i]$.countNodes()    ▷ Recursive function call
    return $N$
Subroutine assignExpr($d(v)$, $\varpi$)
    Let $\varsigma$ be a pseudorandom variable following the uniform distribution over $[0, |\langle\mathbf{E}\rangle[a(v_0)]|]$
    if $a(v_0) = 0$ then
        assignTerm()
    else
        $\Sigma \leftarrow \langle\mathbf{E}\rangle[a(v_0)][\varsigma]$
        for $i \leftarrow 0$ to $a(v_0)$ do
            $\mathbf{C}[i] \leftarrow$ Genotype($d(v_0 + 1)$)
            $\mathbf{C}[i]$.expand(($d(v_0) + 1$), $\varpi$)
        end
    end
Subroutine assignTerm()
    Let $\rho$ be a pseudorandom variable following the uniform distribution over $[0.0, 1.0]$
        $\varsigma$ be a pseudorandom variable following the uniform distribution over $[0, |\langle\mathbf{E}\rangle[0][i]|]$
    $\mathbf{C} \leftarrow 0$
...
```

[a] *cf.* **Appendix A**.

[b] The 3-dimensionality of $\langle\mathbf{E}\rangle$ is substantiated by its subsumption of both *mono-* and *multi*-argumented expressions in tandem.

[c] $\langle\mathbf{E}\rangle$-elements must, however, be pertinent to the behaviour of the agent – that is, as complete expressions, they must function as direct instructions to agent. The domain of valid arguments is therefore more restricted than detailed in **Appendix A**.

```
...
    for $i \leftarrow 0$ to $|\varepsilon_{Term}|$ do
        if $(\rho \leftarrow \rho - p_{Term}[i]) \leq 0$ then $\Sigma[0] \leftarrow \langle \mathbf{E} \rangle[0][i][\varsigma]$
    end
    if $\rho > 0$ then sanitise $\rho$, $\Sigma[0] \leftarrow \rho$
Subroutine getMaxNode()
    $\max[d(v)] \leftarrow d(v_0)$
    for $i \leftarrow 0$ to $a(v_0)$ do $\max[d(v)] \leftarrow \max[\mathbf{C}[i]$.getMaxNode(), $\max[d(v)]]$
    return $\max[d(v)]$
Subroutine getMinTerm()
    if terminal($v_0$) = true then
        return $d(v_0)$
    else
        $\min[d(v):$ terminal$(v)] \leftarrow \max[|\psi(t_0)|] + 1$
        foreach $\psi(t_0) \in \mathbf{C}$ do
            $\min[d(v):$ terminal$(v)] \leftarrow \min[\min[d(v):$ terminal$(v)]$, $\psi(t_0)$.getMinTerm()$]$
        end
        return $\min[d(v):$ terminal$(v)]$
    end
Subroutine setArity($d(v)$)
    Let $\rho$ be a pseudorandom variable following the uniform distribution over $[0.0, 1.0]$
    if $((d(v) > \min|\psi(t_0)|]) \wedge (\rho < 0.35)) \vee (d(v) = \max|\psi(t_0)|])$ then
        $a(v_0) \leftarrow 0$
        terminal($v_0$) $\leftarrow$ true
    else
        Let $\rho'$ be a pseudorandom variable following the uniform distribution over $[0.0, 1.0]$
        terminal($v_0$) $\leftarrow$ false
        for $i \leftarrow 0$ to $|p_{Expr}|$ do
            if $(\rho' = \rho' - p_{Expr}[i]) \leq 0$ then $a(v_0) \leftarrow i + 1$
        end
        if $\rho' > 0$ then $a(v_0) \leftarrow |p_{Expr}|$
        $\mathbf{C} \leftarrow$ Genotype[$a(v_0)$]
    end
Subroutine setDepths($d(v)$)
    $d(v_0) \leftarrow d(v)$
    for $i \leftarrow 0$ to $a(v_0)$ do $\mathbf{C}[i]$.setDepths($d(v) + 1$)
Subroutine insert($\psi(t_0)$)
    define $\lceil \psi(t_0) \rceil \leftarrow \max[|\psi(t_0)|] - (\psi(t_0)$.getMaxNode() $- |\psi(t_0)|)$
           $\lfloor \psi(t_0) \rfloor \leftarrow \max[1,\ (\min[|\psi(t_0)|] - (\psi(t_0)$.getMaxNode() $- |\psi(t_0)|))]$
    Let $\varsigma$ be a pseudorandom variable following the distribution over $[0, (\lceil \psi(t_0) \rceil - \lfloor \psi(t_0) \rfloor) + 1]$
    variable $d'(v) \leftarrow \varsigma + \lfloor \psi(t_0) \rfloor$
    if $(d'(v) + (\psi(t_0)$.getMinTerm() $-|\psi(t_0)|)) - \min[|\psi(t_0)|]$ then
        $d'(v) \leftarrow \min[|\psi(t_0)|] - (\psi(t_0)$.getMinTerm() $- |\psi(t_0)|)$
    end
...
```

```
...
    if (return getMaxNode()) $< d'(v)$ then
        insertAt($\psi(t_0)$, getMaxNode())
    else
        insertAt($\psi(t_0)$, $d'(v)$)
    end
Subroutine insertAt($\psi(t_0)$, $d'(v)$)
    if $d(v_0) = d'(v)$ then
        replace($\psi(t_0)$)
    else
        list $\mathfrak{G} \leftarrow \varnothing$
        for $i \leftarrow 0$ to $a(v_0)$ do
            if $\mathbf{C}[i]$.getMaxNode() $\geq d'(v)$ then add $i \rightarrow \mathfrak{G}$
        end
        Let $\varsigma$ be a pseudorandom variable following the distribution over $[0, |\mathfrak{G}|]$
        $b(v) \leftarrow \mathfrak{G}[\varsigma]$
        $\mathbf{C}[b(v)]$.insertAt($\psi(t_0)$, $d'(v)$)
    end
Subroutine replace($\psi(t_0)$)
    $a(v_0) \leftarrow a(\psi(t_0))$[a]
    terminal($v_0$) $\leftarrow$ terminal($\psi(t_0)$)
    $\Sigma \leftarrow \varnothing$, $|\Sigma| = |\psi(t_0).\Sigma|$
    for $i \leftarrow 0$ to $|\Sigma|$ do $\Sigma[i] \leftarrow \psi(t_0).\Sigma[i]$
    if terminal($\psi(t_0)$) = true then
        $\mathbf{C} \leftarrow \varnothing$
    else
        $\mathbf{C} \leftarrow \varnothing$, $|\mathbf{C}| = a(v_0)$
        for $i \leftarrow 0$ to $a(\psi(t_0))$ do
            $\mathbf{C}[i] \leftarrow$ Genotype($d(v_0) + 1$)
            $\mathbf{C}[i]$.replace($\psi(t_0).\mathbf{C}[i]$)
        end
    end
Let $\varsigma$ be a pseudorandom variable following the distribution over $[0, a(v_0)]$
Subroutine mutateTerm()
    if terminal($v_0$) = false then
        $\mathbf{C}[\varsigma]$.mutateTerm()
    else
        assignTerm()
    end
Subroutine mutateFunc()
    Let $\rho$ be a pseudorandom variable following the distribution over $[0.0, 1.0]$
    if $d(v_0) = 0$ then
        $\mathbf{C}[\varsigma]$.mutateFunc()
    else if $((d(v_0) = (\max[|\psi(t_0)|] - 1)) \vee (\rho < 0.3)$ then
        individual $\varphi(t_0) \leftarrow$ Genotype($d(v_0)$)
...
```

[a] Defined in the usual manner: $a(\psi(t_0)) = |\{w : w \in V_{\Pi(t_0)} \wedge \psi(t_0) = (\text{1-generation parent}(w))\}|$

```
...
        $\varphi(t_0)$.setArity($d(v_0)$)
        $\varphi(t_0)$.assignExpr($d(v_0)$, 0)
        replace($\varphi(t_0)$)
    end
Subroutine clone()
    $\psi'(t_0) \leftarrow$ Genotype($d(v_0)$, $a(v_0)$, terminal($v_0$))      ▷ Denote the *clone* genotype by $\psi'(t_0)$
    $\psi'(t_0).\Sigma \leftarrow \varnothing,\ |\psi'(t_0).\Sigma| \leftarrow |\Sigma|$
    for $i \leftarrow 0$ to $|\Sigma|$ do  $\psi'(t_0).\Sigma[i] \leftarrow \Sigma[i]$
    if terminal($v_0$) $=$ true then
        $\psi'(t_0).\mathbf{C} \leftarrow \varnothing$
    else
        $\psi'(t_0).\mathbf{C} \leftarrow \varnothing,\ |\psi'(t_0).\mathbf{C}| = |\mathbf{C}|$
        for $i \leftarrow 0$ to $|\mathbf{C}|$ do  $\psi'(t_0).\mathbf{C}[i] \leftarrow \mathbf{C}[i]$.clone()
    end
Subroutine getNodeAtDepth($d'(v)$)
    if $d(v_0) = d'(v)$ then
        return
    else
        list $\mathfrak{G} \leftarrow \varnothing$
        for $i \leftarrow 0$ to $a(v_0)$ do
            if $\mathbf{C}[i]$.getMaxNode() $> d'(v)$ then  add $i \rightarrow \mathfrak{G}$
        end
        Let $\varsigma$ be a pseudorandom variable following the distribution over $[0, |\mathfrak{G}|]$
        $b(v) \leftarrow \mathfrak{G}[\varsigma]$
        return $\mathbf{C}[b(v)]$.getNodeAtDepth($d'(v)$)
    end
Subroutine getSubTree(flag)
    Let $\varsigma$ be a pseudorandom variable following the distribution over $[0, a(v_0)]$
        $\varsigma'$ be a pseudorandom variable following the distribution over $[0, ($getMaxNode()$ - 1)]$
    if flag $=$ true then
        if $a(v_0) = 0$ then
            return clone()
        else
            return $\mathbf{C}[\varsigma]$.getSubTree(true)
        end
    else
        $d' \leftarrow \varsigma' + 1$
        return getNodeAtDepth($d'(v)$)
    end
Output: *Subroutine-contingent.*
```