Evolutionary System Definition Language

Steve Dower[*] Clinton Woodward

Swinburne University of Technology
Melbourne, Australia
November 10, 2010

Abstract

A large proportion of publications in evolutionary computation describe algorithm specialisation and experimentation. These algorithms are variously described using text, tables, flowcharts, functions or pseudocode. However, ambiguity is a common artefact that can limit the efficiency of communication. Evolutionary System Definition Language (ESDL) is a conceptual model and language for describing evolutionary systems efficiently and with reduced ambiguity, including systems with multiple populations and adaptive parameters. ESDL can be machine-interpreted to run systems without requiring a hand-coded implementation.

1 Introduction

Of the myriad of approaches to computational intelligence, evolutionary computation (EC) forms a significant proportion [1–3]. The defining feature of an evolutionary algorithm (EA) is a solution population that, over time, improves as the result of competitive forces. The majority of these algorithms derive from the neo-Darwinian paradigm of biological evolution, notably in the use of concepts and terminology of genotypes, phenotypes and fitness-affected survival. Each step in the evolution of the population is the result of applying a selection and reproduction algorithm to the preceding population.

The generality of the biologically inspired EA model has resulted in a vast number of publications presenting improved or optimised algorithms. These algorithms are variously described using text and tables (as in [4–6]), flowcharts (as in [6–8]), pseudocode (as in [8]) and functions (as in [9]). However, these descriptions are often sufficiently ambiguous that independent reproduction of experiments is not possible without a significant number of correct assumptions. Eiben [10], Peer [11], Ventura [12], Rummler [13] and many others have identified the need for algorithms to be completely shareable and verifiable. Early 20th century philosophers Cohen and Nagel [14] stated, “scientific method ... is concerned with verification”. However, much of the published literature in EC consists of new empirical evidence and little verification of earlier results.

An attempt at reproducing an existing algorithm was made by Painter in [15], where he implemented Grammatical Evolution (GE) using Python based on the original published specification and source code. However, his results showed a much higher rate of premature convergence than that found by the original authors, O’Neill and Ryan [16]. Painter attributed his lack of success to incomplete access to the original authors’ source code, particularly that it “omitted the genetic algorithm portion” and “did not compile on its own,” and cited two other similarly failing attempts to implement GE.

Some systems (for example, that shown in Excerpt 2) are not easily implementable in existing evolutionary computation frameworks such as EO,[1] ECJ[2] or CILib.[3] While these frameworks provide

[*]Contact via `http://stevedower.id.au/`

[1]An evolutionary computation framework for C++, online at `http://eodev.sourceforge.net/`

[2]A research evolutionary computation system for Java, online at `http://cs.gmu.edu/~eclab/projects/ecj/`

[3]A component based framework for developing Computational Intelligence software in Java, online at `http://www.cilib.net/`

flexible object models that generally support complex systems, the effort of translating an algorithm description into very specific program code or definitions (often bearing little apparent relation to the original) is high. A likely cause of this high cost is the lack of a common structure for describing the parameters and processes underlying the evolutionary system, coupled with the subjectivity of aesthetic presentation. While it is reasonable to assume that authors select a presentation pleasing to them, it is unlikely that the choice is suitable for everyone.

De Jong [9] provided explicit but verbose pseudocode as an introduction to evolutionary systems (Excerpt 1). In their own introductory text, Eiben and Smith [6] summarised their textual descriptions of breeding processes into tables specifying the operation to use at each stage of a predefined sequence (see Table 1).

Authors such as Koza [7] and O'Neill [4] constructed sets of parameters – "tableaux" – as descriptions of various problems and applicable breeding parameters (for example, that shown in Table 2). While sufficient for clearly defining the problem space, these tables do not describe the structure of the breeding algorithm; both authors use text and an occasional flowchart for this purpose. Price et al. [8] describe the processes behind Differential Evolution (DE) using text, flowcharts, diagrams (such as Figure 1), pseudocode and various mathematical notations. Implementers of DE must reconcile no less than five separate definitions into actual code.

Each of these examples describes particular algorithms in styles that have a learning curve; to understand the algorithm, the reader must first understand the description. Any misunderstanding in either the form of expression or the algorithm itself may result in an incorrect implementation, which, however unfairly, reflects poorly on the original description. An oft-suggested solution is for all researchers to standardise on a single software library, typically the library being promoted by the author. (For examples, see [17–20], as well as those cited previously.) Standardising particular software implementations is very rarely done, largely due to difficulties with portability, licensing and personal opinions. Typically, interchange formats and communication protocols are selected instead. For example, there is no "standard" internet browser: there are standards that describe how a (conforming) browser should communicate, interpret and present information.

An alternative approach, presented here, is to create a domain-specific language that unambiguously specifies the breeding procedure of an evolutionary system without enforcing a particular software framework. Such a description language should support parameterised selection, recombination and mutation operators, dynamic problem spaces, adaptive parameters and distributed populations to be at least capable of expressing current algorithms without hindering the expression of future developments. The use of such a standard form of description would allow researchers to communicate their intent clearly when describing algorithms and the learning curve is diminished or removed.

VHDL[4] is an example of a mature and successful domain-specific language from the digital electronics field [21]. While originally intended as a description language for interoperability, the creation of compilers and synthesisers has transformed it into a development language. However, VHDL also illustrates the necessity of supporting domain knowledge among practitioners; for all its precision, VHDL can be close to impenetrable for those without any experience in digital electronics.

Previous domain-specific modelling languages for evolutionary computation include Evolutionary Algorithm Modeling Language (EAML) [22] and Programmable Parameter Control for Evolutionary Algorithms (PPCEA) [23], neither of which has achieved wide use. EAML represented the breeding process of evolutionary systems primarily for communication and interoperation between distinct frameworks [24]. PPCEA is a scripting language that provides parameter control of an algorithm, without emphasising the structure of the algorithm itself.

Once a description language for the EC domain is developed, a reasonable next step is to create software that can interpret the description language and perform the algorithm directly, removing the manual translation step and changing the "description" language into a "definition" language (as occurred with VHDL). The benefits of an automatic translation from a description language to an executable program are most significant during design, where rapid modification and evaluation is

[4]VHDL: VHSIC hardware description language; VHSIC: very high speed integrated circuit

important. This simplifies independent confirmation of results.

This paper is structured as follows: In Section 2, the conceptual model and syntax of Evolutionary System Definition Language (ESDL) is described. Section 3 contains a set of example system definitions to illustrate the suitability and flexibility of ESDL. Section 4 discusses the design rationale and potential future directions of ESDL.

2 Language

2.1 Overview

ESDL is a domain-specific modelling language for describing the flow of an EA. It does this by defining the *initialisation*, *combination* and *breeding* aspects of an evolutionary system. It does not attempt to define specifics of any particular operator, representation or problem; ESDL depends on an underlying evolutionary computation framework to provide problem landscapes and operator implementations.

The central conceptual entities of ESDL are *individuals* and *groups*. Each individual represents a single solution or part of a solution to a problem being addressed. The only required characteristic of individuals is that they are orderable, that is, two individuals can be compared and determined to be more, less or equally suitable ("fit") than each other.

Groups are collections of multiple individuals and are either *persistent* or *transient*. For example, groups representing the primary populations are generally persistent, while intermediate selections of individuals are considered transient. ESDL does not explicitly distinguish between persistent and transient groups; the distinction represents the intent and use of a particular group.

An individual may be a member of multiple groups simultaneously and appear multiple times in a single group. Hence, the groups that claim an individual do not define the identity of that individual; an individual cannot say that it "belongs" exclusively to a particular generation because the same individual may also have "belonged" to an earlier generation (for example, through elitism) as well as being included in one or more transient groups (for example, as a potential replacement).

2.2 Basic Operations

The primary operation specified by ESDL is *selection*. Using the `FROM–SELECT` statement,[5] individuals are taken from other groups and filtered, combined or otherwise modified to form a new group.

Listing 1: Example of FROM–SELECT in ESDL

```
FROM population SELECT 100 offspring USING tournament(k=2), random_mutate
```

The statement shown in Listing 1 creates a group called `offspring` containing one hundred individuals derived from the `population` group. Individuals are selected using a binary tournament process and then cloned and mutated. It is the underlying implementation that provides the actual behaviour of `tournament` and `random_mutate`; ESDL's only requirement is that they must accept named parameters (where appropriate), such as the probability of mutating a gene, or, as in Listing 1, the tournament size k.

New individuals are created using *generators* that provide an infinite number of individuals. Properties such as genome length and gene bounds are passed to the generator and associated with/retained by each individual so that they do not need to be respecified for each operator. For example, the `random_binary` generator shown in Listing 2 is provided by the underlying implementation and the length of each individual is specified as a parameter.

[5] While bearing some syntactical similarity to SQL, ESDL is quite different; direct comparison to SQL is not relevant and generally unhelpful.

Listing 2: Example of population creation using a generator in ESDL

```
FROM random_binary(length=6) SELECT 50 population
```

Specifying multiple source or destination groups allows for merging and partitioning. Line 1 of Listing 3 splits the `population` group into three groups: ten individuals into `parentsA`, five into `parentsB` and the remainder into `offspring`. Line 2 merges the three groups back into a single group named `everyone`.

Listing 3: Example of group partitioning and merging in ESDL

```
1 FROM pop SELECT 10 parentsA, 5 parentsB, offspring
2 FROM parentsA, parentsB, offspring SELECT everyone
```

Note that the order of selection has not been specified on line 1 of Listing 3. The selection method in this case is to take individuals from groups in the same order they were selected into the groups (first-in-first-out). If ordering is important, it is necessary to specify a filter, such as tournament selection. Selection operators with replacement produce unbounded groups that require the destination to have a size specified, such as `parentsA and parentsB`. If a size were also specified for `offspring` it would be possible to use selection with replacement.

Fitness evaluations are performed by *evaluators*, which are not directly coupled to individuals and their representations. The underlying implementation determines whether the fitness evaluation occurs immediately or lazily;[6] ESDL only requires that the fitness is available when needed. In many cases, a default evaluator may be specified separately from the system definition, allowing the definition to omit details of the evaluator. However, systems that use parameterised evaluators, multi-step evaluations or perform credit-assignment require explicit specification.

Dynamic environments must recalculate fitness values each time the environment updates. The `EVALUATE` command (which may be abbreviated to `EVAL`) instructs all individuals in the given group or groups to recalculate their fitness using a specified evaluator. Listing 4 is an example of a time-based dynamic landscape to evaluate the entire population.

Listing 4: Example of a dynamic evaluator in ESDL

```
t = t + 1
EVAL population USING landscape(time=t)
```

Our current work has shown that many breeding systems are fully describable using only `FROM`–`SELECT` statements (as shown in listings 12–15 and 17). However, complex mutation or coevolution systems can require more complicated functionality than `FROM`–`SELECT` can achieve. For example, DE uses three different individuals from the same group to perform a single mutation operation [8]. This collation of individuals may be expressed in ESDL using the `JOIN` statement, as shown in Listing 5. (An alternative implementation may choose to handle the collation within the `mutate_DE` operation.)

Listing 5: Example of DE-style collation in ESDL

```
JOIN popA, popA, popA INTO parents USING random_tuples(unique=True)
FROM parents SELECT offspring USING mutate_DE
```

The `parents` group in Listing 5 will contain a set of joined individuals. Joined individuals are functionally identical to regular individuals: their "species"[7] specifies that they consist of other individuals, similar to a numeric vector that consists of scalar values.

Coevolutionary systems [1] (both cooperative and competitive) and specific credit-assignment

[6]"Lazy" evaluation retains the calculation required but defers it until the result is needed. If the result is never needed, the calculation never occurs and no computation time is wasted.

[7]There are various definitions of species within the EC field, and while important to the underlying implementation and species-specific operators, ESDL does not require a strict definition.

schemes are realisable using `JOIN` with specially designed evaluators. Joined individuals can be assigned a fitness evaluator, as with separate individuals, however the default evaluator mentioned earlier does not automatically apply. The example of Listing 6 uses Potter and De Jong's CCGA-1 model [25] of joining each individual of one group with the best individual of a second group, then transferring part or all of the fitness value from `joined` to the original individuals in `popA`. (The complete example is shown in excerpts 3–5 and Listing 18.)

Listing 6: A two-step evaluator example using joined individuals

```
JOIN popA, popB INTO joined USING each_with_best
EVALUATE joined USING rastrigin
EVALUATE popA USING assign(source=joined)
```

The first `EVALUATE` statement assumes the existence of a `rastrigin` evaluator that expects joined individuals rather than a single individual. The second `EVALUATE` statement specifies the `assign` evaluator with each individual in the `joined` group to allocate the fitness to the original individual in `popA`. ESDL is agnostic to the type or structure of fitness values and hence credit assignment evaluators are algorithm or problem specific.

2.3 System Structure

A system definition starts with an *initialisation* block where any persistent populations are created and, if necessary, assigned an evaluator. This operation is specified separately to per-generational breeding to allow for systems where a meaningful initial evaluation must be specified explicitly. The first two lines of Listing 7 make up the initialisation block. To specify a non-default evaluator for the initial population, an `EVALUATE` statement would be inserted between lines 1 and 2.

The *generation* block is enclosed by `BEGIN GENERATION` and `END GENERATION` statements. The code within the generation block executes once per generation (or generation-equivalent)[8] and represents the main breeding process of an algorithm. The `REPEAT` and `END REPEAT` statements (lines 5 and 16) simplify gap models [26, 27] that run multiple times per generation-equivalent.

Since ESDL does not distinguish between transient and persistent groups, the `YIELD` command is necessary to identify groups that are relevant in terms of statistics and termination conditions. The `YIELD` statement passes entire groups to an external monitoring system.[9] The group that contains the final solution (in Listing 7, `population`) should always be yielded, but some other groups may also be relevant. For example, statistics collected from the `offspring` and `replacee` groups may show improvements in each breeding operation; many algorithms perform parameter adjustment based on this type of information.

Listing 7 contains a number of `YIELD` statements, returning the initial population (line 2), the updated population each generation-equivalent (line 18) and the offspring and replacee groups each time they are updated (line 13).

[8]"Generation-equivalent" is the preferred term in some works dealing with steady-state algorithms.

[9]`YIELD` behaves like the `yield` statement in the Python and C# programming languages. The specified groups are returned without interrupting the code sequence, similar to the way that a `print` statement displays a value and continues and unlike a `return` statement, which exits a function.

Listing 7: A complete steady-state EA system in ESDL

```
FROM random_real SELECT 500 population
YIELD population

BEGIN GENERATION
  REPEAT 500
    FROM population SELECT 2 parents USING binary_tournament
    FROM parents    SELECT offspring USING crossover(per_pair_rate=0.9), \
                                           mutate(per_gene_rate=0.01)

    FROM offspring  SELECT 1 replacer USING best
    FROM population SELECT 1 replacee, rest USING uniform_random_no_replacement

    YIELD offspring, replacee

    FROM replacer, rest SELECT population
  END REPEAT

  YIELD population
END GENERATION
```

2.4 Adaptive Systems

In order to support flexible parameterisation and adaptive systems, ESDL provides a basic variable/arithmetic system. Variables are created automatically by assignment and may be used throughout the system definition (as shown in Listing 8). Basic arithmetic (addition, subtraction, multiplication and division) and calls to external functions are the minimum requirements of an ESDL implementation. Complicated control-flow structures should be written as external functions.

Listing 8: Examples of assignment in ESDL

```
1 size = 100
2 FROM random_int SELECT (size) population
3 FROM population SELECT (size/10) parents
4
5 mutate_rate = mutate_rate * adapt_rate(population)
```

Surrounding variables or expressions by parentheses (as in lines 2 and 3 of Listing 8) is suggested to distinguish numeric values from group names. Listings 19 and 20 show a complete example using adaptive mutation and external functions.

Assigning a group to a variable creates an alias. In Listing 9 this results in both `parents` and `destination_group` referring to the same group. Assigning another group to an existing alias changes the alias, rather than modifying the group.[10]

Listing 9: Example of group aliasing in ESDL

```
destination_group = parents
FROM population SELECT 2 destination_group
FROM parents SELECT 2 offspring
```

2.5 Syntactical Elements

ESDL is case-insensitive with respect to keywords, group names, variable names and parameter names. Structure is determined by `BEGIN`, `REPEAT` and `END` statements, rather than punctuation or spacing.

[10]`FROM`–`SELECT` must be used to change a group's composition.

Lines beginning with or containing a hash symbol (`#`), double-slash (`//`) or semicolon (`;`) are comments which are ignored from the comment start until the end of the line. The backslash character (`\`) acts as a line-continuation marker, merging the current line with the following. Comments on a line ending with a backslash must be placed after the backslash character.

Lines beginning with a backtick (`` ` ``, also called a grave accent) are assumed to contain non-standard functionality. For example, an implementation producing Python code may choose to pass all lines beginning with a backtick to the interpreter unmodified, such as the example in Listing 10, which allows native Python features and libraries to be readily accessed.

Listing 10: Using backticked lines for Python code

```
`from random import gauss
`mutate_rate *= gauss(0.5, 0.2)
FROM parents SELECT offspring USING mutate(per_gene_rate=mutate_rate)
```

Alternatively, an implementation may accept optimisation or compilation hints, as shown in Listing 11.

Listing 11: Using backticked lines as directives

```
`group-type(parents): transient
FROM population SELECT 10 parents USING random
```

Group, variable and parameter names may contain any alphanumeric character or underscores and must begin with an alphabet character. Names beginning with an underscore are reserved for use by the underlying implementation, for example, if extra named variables are generated when compiling to another language.

Some terms are reserved and may not be used as the names of groups or variables. These are `BEGIN`, `END`, `EVAL`, `EVALUATE`, `FROM`, `INTO`, `JOIN`, `REPEAT`, `SELECT`, `USING` and `YIELD`.

These words may be used as parameter names, since in that context they are contained by parentheses. `GENERATION` is not a reserved word, as it only has meaning when used immediately after a `BEGIN` or `END` statement.

3 Examples

Each example presented in this section consists of a published description of a system, followed by a description of the same system using ESDL. Some examples highlight the comparative simplicity or comprehensibility of ESDL, while others represent canonical or regularly cited models.

While comments may be included in ESDL definitions, these examples avoid their use in order to better demonstrate the readability of ESDL without supporting text. In some examples, Python code or pseudocode is used to describe a particular operator. This demonstrates the abstraction of operator implementations, one of the strengths of ESDL.

3.1 EV Algorithm

De Jong [9] described the simplistic EV algorithm as an introduction to evolutionary systems. Excerpt 1 and Listing 12 show his elaborated EV algorithm in its original form and as an ESDL definition, respectively. The variable M, common to both descriptions, requires a numeric value before the algorithm can be used.

Excerpt 1: De Jong's elaborated EV algorithm [9]

```
EV:
    Randomly generate the initial population of M individuals
    (using a uniform probability distribution over the entire
    geno/phenospace) and compute the fitness of each individual.

    Do Forever:

        Choose a parent as follows:
         - select a parent randomly using a uniform probability
           distribution over the current population

        Use the selected parent to produce a single offspring by:
        - making an identical copy of the parent, and then
          probabilistically mutating it to produce the offspring.

        Compute the fitness of the offspring.

        Select a member of the population to die by:
        - randomly selecting a candidate for deletion from the
          current population using a uniform probability
          distribution; and keeping either the candidate or the
          offspring depending on which one has higher fitness.

    End Do
```

Listing 12: De Jong's EV expressed using ESDL

```
FROM random_individual SELECT (M) population
YIELD population

BEGIN GENERATION
    FROM population SELECT 1 parent USING uniform_random

    FROM parent SELECT offspring USING mutate_random

    FROM population SELECT 1 candidate, others USING uniform_shuffle
    FROM candidate, offspring SELECT 1 replacement USING best
    FROM replacement, others SELECT population

    YIELD population
END GENERATION
```

3.2 Binary-valued Evolutionary Algorithm (EA)

Eiben and Smith [6] describe a number of EA configurations using tables of parameters such as Table 1. The equivalent system as an ESDL definition is given in Listing 13. The variable n, common to both descriptions, requires a value before the algorithm can be used. However, the variable p_m is calculated from the value of n in Listing 13. Note that the termination condition is not specified in Listing 13, since it is not a fundamental part of the algorithm itself.

Table 1: Description of the EA for the Knapsack Problem [6]

Representation	Binary strings of length n
Recombination	One point crossover
Recombination probability	70%
Mutation	Each value inverted with independent probability p_m
Mutation probability p_m	$1/n$
Parent selection	Best out of random 2
Survival selection	Generational
Population size	500
Number of offspring	500
Initialisation	Random
Termination condition	No improvement in last 25 generations

Listing 13: The EA in Table 1 expressed using ESDL

```
FROM random_binary(length=n) SELECT 500 population
YIELD population

BEGIN GENERATION
    FROM population SELECT 500 parents USING binary_tournament
    FROM parents SELECT offspring USING crossover_one(per_pair_rate=0.7)
    FROM offspring SELECT offspring USING mutate_bitflip(per_gene_rate=(1.0/n))

    FROM offspring SELECT population

    YIELD population
END GENERATION
```

3.3 Grammatical Evolution (GE)

Parameter tables accompany the GE problems presented by O'Neill and Ryan [4]. The tables primarily specify the problem landscape parameters but also include evolution parameters. Table 2 contains one such row of a table in [4] and Listing 14 shows an ESDL definition using these parameters. As earlier, the termination condition is not part of the system definition in Listing 14.

Table 2: Subset of a GE tableau [4]

Parameters:	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01, Steady State

Listing 14: The GE parameters and implied system of Table 2 expressed using ESDL

```
FROM random_int SELECT 500 population
YIELD population

BEGIN GENERATION
    FROM population SELECT 2 parents USING binary_tournament
    FROM parents    SELECT offspring USING \
                           crossover_one_different(per_pair_rate=0.9), \
                           mutate_random(per_gene_rate=0.01), \
                           mutate_duplicate(per_indiv_rate=0.01)

    FROM offspring  SELECT 1 replacer USING best
    FROM population, offspring SELECT 500 population USING best

    YIELD population
END GENERATION
```

3.4 Generalized Generation Gap (G3) Model

Deb et al. [5] describe a system for real parameter optimisation, shown in Excerpt 2. Listing 15 shows the equivalent system expressed using ESDL.

Excerpt 2: The G3 system described in [5]

1. From the population P, select μ parents randomly.
2. Generate λ offspring from μ parents using a recombination scheme.
3. Choose two parents at random from the population P.
4. Of these two parents, one is replaced with the best of λ offspring and the other is replaced with a solution chosen by a roulette-wheel selection procedure from a combined population of λ offspring and two chosen parents.

Listing 15: The G3 system in Excerpt 2 expressed using ESDL

```
FROM random_real SELECT 100 population
YIELD population

BEGIN GENERATION
    FROM population SELECT (mu) parents USING uniform_random
    FROM parents SELECT (lambda) offspring USING crossover

    FROM population SELECT 2 parents, remainder USING uniform_shuffle
    FROM offspring SELECT 1 replacement_1 USING best
    FROM parents, offspring SELECT 1 replacement_2 USING fitness_proportional

    FROM remainder, replacement_1, replacement_2 SELECT population
    YIELD population
END GENERATION
```

3.5 Genetic Programming (GP) using ECJ

The ECJ software package uses a linear notation[11] to describe evolutionary systems. The system described by Koza [7] is partially[12] shown using ECJ's notation in Listing 16 and as an ESDL definition in Listing 17. (Some parameters used in Listing 17 are not shown in Listing 16.)

Listing 16: Koza's GP system [7] partially expressed using ECJ's parameter format

```
pop.subpop.0.size = 1000
pop.subpop.0.species = ec.gp.GPSpecies
pop.subpop.0.species.ind = ec.gp.GPIndividual
pop.subpop.0.species.ind.numtrees = 1
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.0.tc = tc0
pop.subpop.0.species.numpipes = 2
pop.subpop.0.species.pipe.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.0.prob = 0.9
pop.subpop.0.species.pipe.1 = ec.gp.koza.ReproductionPipeline
pop.subpop.0.species.pipe.1.prob = 0.1
```

Listing 17: Koza's GP system [7] expressed using ESDL

```
FROM real_tgp(terminal_prob=0.1,deepest=6,terminals=1) SELECT 1000 population
YIELD population

BEGIN GENERATION
    FROM population SELECT 100 reproduced, 900 parents USING tournament(k=7)
    FROM parents SELECT offspring USING crossover_one(deepest_result=17)

    FROM reproduced, offspring SELECT population
    YIELD population
END GENERATION
```

3.6 Cooperative Coevolutionary Genetic Algorithm 1 (CCGA-1)

Potter and De Jong in [25] describe and demonstrate CCGA-1 as a "starting point" model for cooperative coevolution. Excerpts 3–5 show the original descriptions from [25] and Listing 18 shows the equivalent ESDL description. The ESDL description is somewhat verbose since many similar lines

[11]The notation is Java's property list format, which uses a `key=value` notation.
[12]The complete description is over 150 lines and is available at http://cs.gmu.edu/~eclab/projects/ecj/docs/parameters.html

are repeated for each population. ESDL deliberately omits a method for generalising to multiple populations, as discussed later.

Excerpt 3: CCGA-1 algorithm from [25]

```
gen=0
for each species $s$ do begin
    $Pop_s(gen)$ = randomly initialized population
    evaluate fitness of each individual in $Pop_s(gen)$
    end
while termination condition = false do begin
    $gen = gen + 1$
    for each species $s$ do begin
        select $Pop_s(gen)$ from $Pop_s(gen–1)$ based on fitness
        apply genetic operators to $Pop_s(gen)$
        evaluate fitness of each individual in $Pop_s(gen)$
        end
    end
```

Excerpt 4: CCGA-1 fitness evaluation description from [25]

CCGA-1 begins by initializing a separate population of individuals for each function variable. The initial fitness of each subpopulation member is computed by combining it with a random individual from each of the other species and applying the resulting vector of variable values to the target function.

After this startup phase, each of the individual subpopulations in CCGA-1 is coevolved in a round-robin fashion using a traditional GA. The fitness of a subpopulation member is obtained by combining it with *the current best* subcomponents of the remaining (temporarily frozen) subpopulations.

Excerpt 5: CCGA-1 experimental parameters from [25]

representation:	binary (16 bits per function variable)
selection:	fitness proportionate
fitness scaling:	scaling window technique (width of 5)
elitist strategy:	single copy of best individual preserved
genetic operators:	two-point crossover and bit-flip mutation
mutation probability:	1/chromlength
crossover probability:	0.6
population size:	100

Listing 18: CCGA-1 expressed for two populations using ESDL

```
FROM random_binary(length=16) SELECT 100 popA, 100 popB

JOIN popA, popB INTO joinedA USING each_with_random
EVALUATE joinedA USING rastrigin
EVALUATE popA USING assign(source=joinedA)

JOIN popB, popA INTO joinedB USING each_with_random
EVALUATE joinedB USING rastrigin
EVALUATE popB USING assign(source=joinedB)

YIELD joinedA, joinedB, popA, popB

BEGIN GENERATION
    FROM popA SELECT parents USING fitness_proportional
    FROM popA SELECT 1 elitist USING best
    FROM parents SELECT offspring USING crossover_two(per_pair_rate=0.6), \
                                        mutate_bitflip(per_gene_rate=1/16)
    FROM offspring, elitist SELECT 100 popA USING best

    JOIN popA, popB INTO joinedA USING each_with_best
    EVALUATE joinedA USING rastrigin
    EVALUATE popA USING assign(source=joinedA)
    YIELD joinedA, popA

    FROM popB SELECT parents USING fitness_proportional
    FROM popB SELECT 1 elitist USING best
    FROM parents SELECT offspring USING crossover_two(per_pair_rate=0.6), \
                                        mutate_bitflip(per_gene_rate=1/16)
    FROM offspring, elitistB SELECT 100 popB USING best

    JOIN popB, popA INTO joinedB USING each_with_best
    EVALUATE joinedB USING rastrigin
    EVALUATE popB USING assign(source=joinedB)
    YIELD joinedB, popB
END GENERATION
```

3.7 Evolution Strategies (ES)

Eiben and Smith describe ES in text form, including the 1/5th adaptive mutation rule [6]. Listings 19 and 20 show ES using an ESDL definition to describe the groups and their interactions and pseudocode to describe the statistical calculations and conditional adaptation.

Listing 19: Pseudocode for the ES functions used in Listing 20

```
function calculate_success_rate(parents, offspring)
    # Return the percentage of offspring who are more fit than their parent.
    count = 0

    for each parent and offspring in parents and offspring
        if offspring.fitness > parent.fitness then increment count

    return count / number of parents


function adapt(current_step, adapt_step, success_rate)
    # Use the 1/5th adaptive mutation rule to return the new step size.
    new_step = current_step

    if success_rate > 0.21:
        new_step = current_step * (1.0 + adapt_step)
    otherwise, if success_rate < 0.19:
        new_step = current_step * (1.0 - adapt_step)

    return new_step
```

Listing 20: ES expressed using ESDL

```
FROM random_real SELECT 10 population
YIELD population

step_size = 1.0
adapt_step_size = 0.1

BEGIN GENERATION
    FROM population SELECT 1 parent USING uniform_random
    FROM parent SELECT 10 parents USING repeat
    FROM parents SELECT offspring USING \
         mutate_gaussian(step_size=current_step, \
         per_gene_rate=1.0)
    YIELD offspring

    # Adjust the step size based on the success rate offspring.
    # (calculate_success_rate and current_step are defined above)
    success_rate = calculate_success_rate(parents, offspring)
    step_size = adapt(step_size, adapt_step_size, success_rate)

    FROM population, offspring SELECT 10 population USING best
    YIELD population
END GENERATION
```

3.8 Differential Evolution (DE)

The DE algorithm [8] is summarised into the flow chart shown in Figure 1. The equivalent ESDL description is given in Listing 21. The weighted-difference-vector mutation operation is abstracted from the system definition, allowing a more suitable programming language be used, such as Python or MATLAB. Listing 22 shows a potential implementation for `mutate_DE` as a generator in Python.

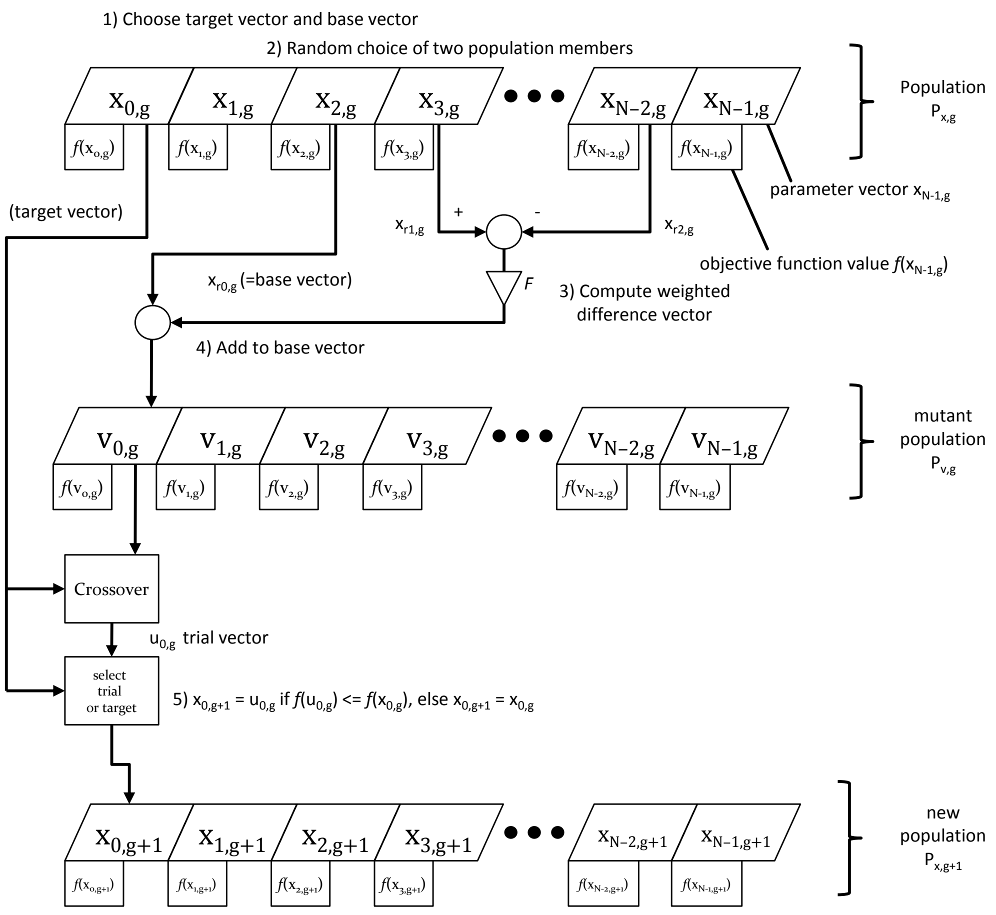
1) Choose target vector and base vector
2) Random choice of two population members
Population $P_{x,g}$
parameter vector $x_{N-1,g}$
objective function value $f(x_{N-1,g})$
(target vector)
$x_{r1,g}$
$x_{r2,g}$
$x_{r0,g}$ (=base vector)
F
3) Compute weighted difference vector
4) Add to base vector
mutant population $P_{v,g}$
Crossover
$u_{0,g}$ trial vector
select trial or target
5) $x_{0,g+1} = u_{0,g}$ if $f(u_{0,g}) <= f(x_{0,g})$, else $x_{0,g+1} = x_{0,g}$
new population $P_{x,g+1}$

Figure 1: DE's generate-and-test loop from Price et al. [8]

Listing 21: The sequence in Figure 1 expressed using ESDL

```
FROM random_real SELECT 100 population
YIELD population

BEGIN GENERATION
    # SELECT without a USING makes a copy of the group
    FROM population SELECT 100 targets

    # Stochastic Universal Sampling for bases
    FROM population SELECT (size) bases USING fitness_sus(mu=size)

    # Ensure r0 != r1 != r2, but any may equal i
    JOIN bases, population, population INTO mutators \
                                   USING random_tuples(distinct=True)

    # mutate_DE is defined in Listing 22
    FROM mutators SELECT mutants USING mutate_DE(scale=0.8)

    JOIN targets, mutants INTO target_mutant_pairs USING tuples
    FROM target_mutant_pairs SELECT trials \
                             USING crossover_tuple(per_gene_rate=0.8)

    JOIN targets, trials INTO target_trial_pairs USING tuples
    FROM target_trial_pairs SELECT population USING best_of_tuple

    YIELD population
END GENERATION
```

Listing 22: The mutation operator used in Listing 21 expressed using Python

```
def mutate_DE(source, scale):
    '''A generator that yields one mutated genome for every tuple of genomes
    passed in source.'''

    for joined_individual in source:
        base, parameter1, parameter2 = joined_individual[:]
        yield [b + scale * (p1 - p2) for b, p1, p2 in \
               zip(base, parameter1, parameter2)]
```

4 Discussion

4.1 Implementation

An ESDL implementation requires an execution engine and a collection of domain operators. The execution engine runs the ESDL system and may be an interpreter, a compiler or a hybrid of the two. ESDL is easily transformed into other programming languages, particularly those with a well-defined iterator or enumerator pattern. ESDL can also be transformed into the forms used by other frameworks, such as ECJ's parameter format (shown in Listing 16), provided the underlying system is capable of supporting the algorithm.

A collection of basic filtering, joining and modification operators is required, as is the ability to specify custom operators. There is not yet a standard set of operators for ESDL implementations, however, standardising the behaviour of common operators may be beneficial to the portability of system descriptions.

The current reference implementation is `esec`, available from `http://code.google.com/p/esec`. `esec` is a Python based framework that transforms ESDL into Python code and executes it using the Python interpreter.

4.2 Immutable Individuals

Part of the contract between ESDL systems and external operators is that the solution represented by an individual does not change, that is, a given individual always receives the same fitness when evaluated with a static evaluator.

This contract allows individuals to receive different fitness values over dynamic landscapes or where the evaluator changes as the result of an `EVALUATE` statement. However, operators that "modify" individuals (such as mutation or crossover) must create new individuals rather than modifying the originals. This avoids the need for the explicit cloning of individuals required by systems that perform modifications *in situ*.

4.3 Conditional Statements

The lack of conditional statements[13] in ESDL is deliberate. Limiting the number of paths through a system to one significantly reduces the complexity [28]. Systems described using ESDL are effectively fixed networks defining how individuals are routed between various groups.

This omission does not limit the ability of ESDL in terms of describing adaptive systems, as complex operations can and should be described separately in a more appropriate language (as in listings 19–22). However, the potential need is acknowledged, and the authors are currently designing an extension to ESDL to allow networks that are more flexible.

As with conditional statements, the omission of conditional loops is deliberate. While a simple repeated-expansion construct[14] would allow some multi-population systems (such as Listing 18) to be written more tersely, it would be insufficient for systems with any difference between populations. A construct that was sufficient for handling complex cases would greatly reduce the readability of ESDL, while only serving to remove repetition that is readily apparent to the human eye.

4.4 FROM–SELECT Ordering

The keywords in the `FROM–SELECT` statement could have been ordered in a multitude of ways. The selected ordering, "`FROM` source `SELECT` destination `USING` filter," best matches ESDL's model of processing and is consistent with its other instructions.

Exchanging `FROM` and `SELECT` would produce a more grammatically correct statement (in English), for example, "`SELECT` destination `FROM` source `USING` filter." When read, however, this emphasises the

[13]Such as `if` and `while`.

[14]A form of macro that repeats a block of code multiple times as if the developer had done so themselves.

result of the operation over the source. The statement "`SELECT` destination `USING` filter `FROM` source" is similar to the mathematical function notation ($dest = f\,(src)$) but further reduces the importance of the source group. `FROM`–`SELECT`–`USING` best describes where to begin, where to finish and how to get there.

Furthermore, the `FROM`–`SELECT`–`USING` ordering matches "`JOIN` sources `INTO` destination `USING` joiner," which improves the readability of ESDL. The alternative orderings of `JOIN`–`INTO`–`USING` are grammatically clumsy and not considered.

5 Summary

A common language for describing algorithms in evolutionary computation could be beneficial to the field. The present state of the field limits independent verification and potentially prevents algorithms gaining wider use. A standardised description language allows algorithms and structures to be defined clearly, and with sufficient detail, that other researchers are able to confidently implement and make use of them.

We have presented Evolutionary System Definition Language (ESDL), which describes systems in a form that is unambiguous and abstracted from the behaviour and implementation of domain operators. From the examples given, we have demonstrated that ESDL is capable of representing a range of evolutionary algorithms. ESDL emphasises groups over individuals or representations and is concise for simple systems while being sufficient for complex, multiple population systems.

ESDL is suitable for automatic transformation into executable code or input to existing frameworks, while most other forms of description require manual interpretation and translation. By simplifying this translation, algorithms expressed in ESDL are easier to use correctly and are more likely to be used by the wider evolutionary computation community.

References

[1] E. Alba and J. M. Troya, “A survey of parallel distributed genetic algorithms,” *Complexity*, vol. 4, 1999. 1, 4

[2] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, 2007. 1

[3] K. A. De Jong, “Evolutionary Computation,” 2009. 1

[4] M. O’Neill and C. Ryan, *Grammatical Evolution.* Kluwer Academic Publishers, 2003. 1, 2, 11

[5] K. Deb, A. Anand, and D. Joshi, “A computationally efficient evolutionary algorithm for real-parameter optimization,” *Evolutionary Computation*, vol. 10, December 2002. 1, 11

[6] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing.* Springer, 2003. 1, 2, 10, 15

[7] J. R. Koza, *Genetic Programming: On The Programming of Computer Programs by Natural Selection.* MIT Press, 1992. 1, 2, 12

[8] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution.* Springer, 2005. 1, 2, 4, 16

[9] K. A. De Jong, *Evolutionary Computation: A Unified Approach.* MIT Press, 2006. 1, 2, 8

[10] A. E. Eiben and M. Jelasity, “A critical note on experimental research methodology in EC,” in *Proceedings of the 2002 Congress on Evolutionary Computation.* IEEE Press, 2002. 1

[11] E. S. Peer, A. P. Engelbrecht, and F. van den Bergh, “Building sustainable collaborative research software.” 1

[12] S. Ventura, C. Romero, A. Zafra, J. Delgado, and C. Hervás, “JCLEC: a Java framework for evolutionary computation,” *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, vol. 12, February 2008. 1

[13] A. Rummler and T. Strufe, “Evolvica - a framework for evolutionary computation,” 2004. 1

[14] M. R. Cohen and E. Nagel, *Introduction to Logic and Scientific Method.* Routledge & Kegan Paul Ltd, 1934. 1

[15] T. Painter, “Grammatical Evolution in Python,” 2006. 1

[16] M. O’Neill and C. Ryan, “Grammatical Evolution: A steady state approach,” in *Late Breaking Papers at the Genetic Programming 1998 Conference.* Omni Press, 1998. 1

[17] J. Alcalá-Fernández, L. Sánchez, S. García, M. J. del Jesus, S. Ventura, J. M. Garrell, J. Otero, C. Romero, J. Bacardit, V. M. Rivas, J. C. Fernández, and F. Herrera, “KEEL: a software tool to assess evolutionary algorithms for data mining problems,” *Soft Computing*, vol. 13, February 2009. 2

[18] J. J. Merelo and A. Prieto, “GAGS, a flexible object-oriented library for evolutionary computation,” in *Proceedings of the First International Workshop on Machine Learning, Forecasting and Optimization*, 1996. 2

[19] J. J. Merelo, P. A. Castillo, and E. Alba, “Algorithm::Evolutionary, a flexible Perl module for evolutionary computation,” *Soft Computing*, vol. 14, 2010. 2

[20] J.-B. Mouret and S. Doncieux, “Sferesv2: Evolvin’ in the multi-core world,” in *Proceedings of the 10th International Congress on Evolutionary Computation.* IEEE Computer Society, 2010. 2

[21] P. J. Ashenden, *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, 1995. 2

[22] C. B. Veenhuis, K. Franke, and M. Köppen, "A semantic model for evolutionary computation," in *6th International Conference on Soft Computing*, 2000. 2

[23] S.-H. Liu, M. Mernik, and B. R. Bryant, "Parameter control in evolutionary algorithms by domain-specific scripting language PPCEA," in *Proceedings of the 1st International Conference on Bioinspired Optimization Methods and their Applications*, 2004. 2

[24] M. Nowostawski, "eaml-design mailing list," February 2002, http://sourceforge.net/mailarchive/forum.php?thread_name=3C72DA51.3030709%40marni.otago.ac.nz&forum_name=eaml-design. 2

[25] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *Proceedings of the The Third Conference on Parallel Problem Solving from Nature*, 1994. 5, 12, 13

[26] G. Syswerda, "A study of reproduction in generational and steady-state genetic algorithms," in *Foundations of Genetic Algorithms.* Morgan Kaufmann Publishers, 1991. 5

[27] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, University of Michigan, 1975. 5

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, December 1976. 18