

Available from: http://doi.acm.org/10.1145/2001576.2001718

Copyright © ACM, 2011.

The definitive version was published in Proceedings of GECCO, (2011).

This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. If your library has a subscription to these conference proceedings, you may also be able to access the published version via the library catalogue.
ESDL: A Simple Description Language for Population-Based Evolutionary Computation

Steve Dower∗ Clinton J. Woodward
April 7, 2011

Abstract

A large proportion of publications in the field of evolutionary computation describe algorithm specialisation and experimentation. Algorithms are variously described using text, tables, flowcharts, functions or pseudocode. However, ambiguity that can limit the efficiency of communication is common. 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 may also be machine-interpreted, allowing algorithms to be tested without requiring a hand-coded implementation, as may already be done using the esec framework. The style is distinct from existing notations used within the field and is easily recognisable. This paper describes the case for ESDL, provides an overview of ESDL and examples of its use.

1 Introduction

Evolutionary algorithms (EA) form a significant proportion of the myriad of approaches to computational intelligence [1, 7, 17]. The defining feature of an EA is a solution population that, over time, improves as the result of competitive forces. Most EAs 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 a 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 [8, 15, 27]), flowcharts (as in [15, 19, 31]), pseudocode (as in [31]) and functions (as in [6]). However, these descriptions often contain ambiguous components that prevent independent reproduction of experiments without a significant number of correct assumptions. Eiben [14], Peer [29], Ventura [35], Rummler [32] and many others have identified the need for algorithms to be completely shareable and verifiable. Jon Claerbout coined the term “reproducible research” to refer to this need [18] and early 20th century philosophers Cohen and Nagel stated, “scientific method ... is concerned with verification” [4]. However, much of the published literature in EC consists of new empirical evidence but little verification of earlier results. While “reproducibility” has been gaining use as a review criterion for conferences, under review conditions and constraints it can be difficult to evaluate.

An oft-suggested solution is for researchers to standardise on a single implementation of a software library or framework, typically the one being promoted by the author. (For examples, see [2, 22–24]) Standardising implementations is generally uncommon, due to difficulties with portability, versioning, licensing and the general inclination of computer scientists to write their own software. Typically, common interchange formats and communication protocols are designed instead. For example, while there is no “standard” internet browser, there are standards that describe how a conforming browser should communicate, interpret and present information.

Some systems (for example, that shown in Excerpt 1) cannot be easily implemented in existing frameworks such as EO,1 ECJ2 or CILib.3 While these frameworks provide flexible object models that generally support complex systems, significant effort is required to translate an algorithm description into executable code. 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 reason-

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/
able to assume that authors select a presentation pleasing to them, it is unlikely that the choice is ideal for the full range of readers.

De Jong provided unambiguous yet verbose pseudocode as an introduction to evolutionary systems [6]. In their own introductory text, Eiben and Smith summarised their textual descriptions of breeding processes into tables specifying the operation to use at each stage of a predefined sequence (see Table 1, from [15]). Some authors including Koza [19] and O’Neill [27] have constructed tables of parameters – “tableaux” – as descriptions of various problems and applicable breeding parameters. 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. describe the processes behind Differential Evolution (DE) using text, flowcharts, diagrams, pseudocode and various mathematical notations [31]. Implementers of DE must reconcile no less than five separate definitions into a working algorithm.

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 interpretation that, however unfairly, reflects poorly on the original description.

An attempt to reproduce an existing algorithm was made by Painter, who implemented Grammatical Evolution (GE) using Python based on the original published specification and source code [28]. However, his results showed a much higher rate of premature convergence than that found by the original authors, O’Neill and Ryan [26]. 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.

Rather than standardising on a specific piece of software, the approach suggested in this work is to specify a domain-specific language suitable for describing the breeding procedure of an evolutionary system unambiguously and 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 limiting future developments. A common form of description would allow researchers to communicate their intent clearly when describing algorithms and diminish or remove the learning curve.

Previous domain-specific modelling languages for EC include Evolutionary Algorithm Modeling Language (EAML) [34] and Programmable Parameter Control for Evolutionary Algorithms (PPCEA) [20], neither of which has achieved wide use. EAML represents the breeding process of evolutionary systems using XML for communication and interoperability between distinct frameworks [25]. PPCEA is a scripting language that provides parameter control of an algorithm, without emphasising the structure of the algorithm itself. Both EAML and PPCEA emphasise implementation and are designed for explaining an algorithm to a machine, rather than other researchers.

VHDL4 is an example of a mature and successful domain-specific language from the digital electronics field [3]. While originally intended as a description language for communication and interoperability, the creation of compilers and synthesisers has transformed it into a development language. However, VHDL also illustrates the need for practitioners to have strong domain knowledge; for all its precision, VHDL can be close to impenetrable for those without any experience in digital electronics. Following this lead, 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.

The benefits of an automatic translation from a description language to an executable program are most significant during design, where rapid modification and evaluation are important, and later, to simplify re-use of an algorithm and independent confirmation of results. (These two stages have already been implemented for the language described in this paper as the esec framework5 and the esecui prototyping tool.6) However, the most interested readers are human, and hence the language must be designed for human readership and not to simplify machine interpretation.

This paper presents and describes Evolutionary System Definition Language (ESDL). The structure of the paper is as follows: In Section 2, the conceptual model and syntax of ESDL is described. Section 3 contains a set of example system definitions to illustrate the suit-

---

4VHDL: VHSIC hardware description language; VHSIC: very high speed integrated circuit
5An evolutionary computation framework for Python, online at http://esec.googlecode.com/
6An algorithm prototyping tool based on esec, online at http://esecui.googlecode.com/
2 Evolutionary System Definition Language

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 EC framework to provide problem landscapes and operator implementations. Algorithms written in ESDL may be transferred between various frameworks, given consistent operators. The specification of some form of “standard library” is essential in the long term, but is beyond the scope of this paper.

The central conceptual entities of ESDL are individuals, groups and operators. Each individual represents a single solution or part of a solution to a problem being addressed and a group represents a collection of individuals. Operators include filters and selectors, which are used to create subsets of groups, and variation operators, which create modified individuals. The flow of individuals through a set of operators is what defines the behaviour of a system.

A group is defined as a list (strictly, an ordered multiset) of some or all of the individuals that have been created at any point during the algorithm’s execution, where the size of a group is the number of individuals in the list including repetitions (that is, the group’s cardinality). Multiple groups may contain the same individual simultaneously and each group may contain repetitions of individuals, and all groups may only contain a finite number of individuals. Groups are named and may be redefined by replacing their contents. An individual’s groups do not define the identity of that individual; an individual cannot say that it “belongs” exclusively to a particular generation’s population 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, after crossover but before mutation).

ESDL is based on plain text, allowing it to be easily presented in publications, typed and modified by hand. The syntax is deliberately distinctive to prevent confusion with existing styles of pseudocode, mathematical notations and equations. Some concepts are similar to those used in functional programming, although ESDL cannot be considered a functional language due to the lack of referential transparency inherent in the stochastic processes used.

The primary statement connecting groups and operators is the FROM–SELECT statement. Individuals are taken from 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 used to derive slightly modified (mutated) copies. Note that it is an underlying implementation that provides the actual behaviour of tournament and random_mutate; ESDL only “glues” pieces of an algorithm together and provides a consistent interface for specifying parameters, such as the probability of mutating a gene, or, as in Listing 1, the tournament size $k$.

New individuals are created using generators that act like groups with an infinite number of individuals. Properties such as genome length and value bounds are passed to the generator and associated with each individual so that they do not need to be explicitly repeated for other operators. Listing 2 uses a random_binary generator to create a group of 50 individuals, each of length six.
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 pop into three groups: ten individuals into parentsA, five into parentsB and the remainder into offspring. Line 2 merges the three groups into a single group named everyone.

Listing 3: Example of group partitioning and merging in ESDL

FROM pop SELECT 10 parentsA, 5 parentsB, offspring
FROM parentsA, parentsB, offspring SELECT everyone

Selecting specific individuals, such as tournament selection, is performed with a selection operator (selector). Selection operators that use replacement produce unbounded groups and require all destination groups to have a size specified, as parentsA and parentsB do (but offspring does not) in Listing 3. If not enough individuals are available to fill a group to its specified size, the result will be smaller than requested. This is useful for algorithms such as CHC, which allows the number of offspring to vary each iteration [16].

Fitness evaluations are performed by evaluators, which are decoupled from individuals and their representations. The underlying implementation determines whether the fitness evaluation occurs immediately or lazily.8 ESDL only requires that the fitness is available when needed, for example, when a fitness proportional selector is used. In many cases, definitions may omit details of the evaluator, allowing application to a range of problems without modification. However, systems that use parameterised evaluators, multi-step evaluations or perform credit-assignment require explicit specification using the EVALUATE (or EVAL) command.

The solutions represented by individuals do not change; a given individual always receives the same fitness when evaluated with a given evaluator. By implication, operators that “modify” individuals (such as mutation or crossover) must create new individuals rather than changing the originals (sometimes called copy-on-write semantics). This avoids the need for the explicit cloning of individuals required by systems that perform modifications in place. Also by implication, dynamic problems (where the fitness of an individual changes with time or some external parameter) cannot be represented as a single evaluator. To solve this limitation, an evaluator’s identity includes its parameters. For example, a dynamic Travelling Salesman Problem evaluator at time \( t_1 \) is not the same as at time \( t_2 \); hence, the fitness of a given individual is not required to remain constant.

The EVALUATE command specifies the evaluator to use for all individuals in the given group or groups. Listing 4 shows an example of using an evaluator parameterised on time.

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 population-based evolutionary systems can be fully described using only FROM–SELECT statements (as shown in listings 9, 10 and 12). Complex mutation schemes and coevolutionary systems can require functionality other than that provided by FROM–SELECT. For example, DE uses three different individuals from the same group to perform a single mutation operation [31]. This collation of individuals may be expressed in ESDL using the JOIN statement, as shown in Listing 5.

Listing 5: Example of DE-style collation in ESDL

JOIN popA, popA, popA INTO parents \\nUSING unique_random_tuples
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” specifies that they consist of other individuals, similar to a numeric vector that consists of scalar values.

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)

Coevolutionary systems (both cooperative and competitive) [1] and specific credit-assignment schemes can be realised using JOIN-INTO with specially designed evaluators. Joined individuals can be evaluated in the same manner as regular individuals, however, fitness is not automatically propagated to the individuals comprising the joined individual. Listing 6 demonstrates Potter and De Jong’s CCGA-1 model [30] 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 [11].)

8Lazy evaluation stores the calculation but does not perform it until the result is needed. If the result is never needed, the calculation never occurs and no computation time is wasted.

9There 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.
The first `EVALUATE` statement assumes a `rastrigin` evaluator that uses a joined individual rather than a single individual with two values. The second `EVALUATE` statement specifies an `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 credit assignment evaluators are necessarily algorithm or problem specific.

To support flexible parameterisation and adaptive systems, ESDL allows variables to be used in place of groups, sizes and parameter values. Variables are created automatically by assignment and support basic arithmetic (addition, subtraction, multiplication and division) and calls to external functions. Conditional control-flow structures are not supported and should be written as external functions.

A complete algorithm expressed using ESDL is called a `system definition`. It consists of an initialisation section where all groups and variables are created; per-iteration breeding is specified separately. The first two lines of Listing 8 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.

`Blocks` represent an iteration of the algorithm. A block is enclosed by `BEGIN` and `END` statements, with the name of the block specified following `BEGIN` (for example, line 4 of Listing 8). The code within a block executes once per iteration and represents the main breeding processes of an algorithm. Multiple blocks (with distinct names) may be specified and executed in an arbitrary order, for example, as a hybrid algorithm [9]. The `REPEAT` statement (line 5) simplifies models that apply variations multiple times per iteration (such as gap or steady-state algorithms) [5,33].

When using an ESDL system definition for executing an algorithm, not all groups created are relevant in terms of statistics or termination conditions. The `YIELD` command identifies those that are relevant by passing entire groups to an external monitoring system. The group that contains the final solution (in Listing 8, `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 resulting from each breeding operation; many algorithms perform parameter adjustment based on this type of information. Termination conditions are not intrinsic to an algorithm, and so are not specified in the system definition.

Listing 8 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).
3.1 Binary-valued Evolutionary Algorithm

Eiben and Smith [15] 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 9. 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 9. Note that the termination condition is not specified in Listing 9, since it is a configuration property rather than a facet of the algorithm.

3.2 Generalized Generation Gap (G3) Model

Excerpt 1: The G3 system described in [8]

1. From the population \( P \), select \( \mu \) parents randomly.
2. Generate \( \lambda \) offspring from \( \mu \) 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 \( \lambda \) offspring and the other is replaced with a solution chosen by a roulette-wheel selection procedure from a combined population of \( \lambda \) offspring and two chosen parents.

Deb et al. [8] describe a system for real parameter optimisation, shown in Excerpt 1. Listing 10 shows the equivalent system expressed using ESDL.

Table 1: Description of the EA for the Knapsack Problem as presented in [15]

<table>
<thead>
<tr>
<th>Representation</th>
<th>Binary strings of length ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recombination</td>
<td>One point crossover</td>
</tr>
<tr>
<td>Recombination probability</td>
<td>70%</td>
</tr>
<tr>
<td>Mutation</td>
<td>Each value inverted with independent probability ( p_m )</td>
</tr>
<tr>
<td>Mutation probability ( p_m )</td>
<td>( 1/n )</td>
</tr>
<tr>
<td>Parent selection</td>
<td>Best out of random 2</td>
</tr>
<tr>
<td>Survival selection</td>
<td>Generational</td>
</tr>
<tr>
<td>Population size</td>
<td>500</td>
</tr>
<tr>
<td>Number of offspring</td>
<td>500</td>
</tr>
<tr>
<td>Initialisation</td>
<td>Random</td>
</tr>
<tr>
<td>Termination condition</td>
<td>No improvement in last 25 generations</td>
</tr>
</tbody>
</table>
3.3 Genetic Programming using ECJ

The parameter files used with the ECJ software package to describe evolutionary systems are based on a linear notation. The system described by Koza is partially shown using ECJ’s notation in Listing 11 and as an ESDL definition in Listing 12. (Some parameters included in Listing 12 have been omitted from Listing 11.)

The ESDL system definition takes eight lines to represent the same algorithm that requires 70 in ECJ’s notation. A complete configuration for esec, including program node specifications and a fitness evaluation function, requires less than fifty lines of code, while the complete ECJ parameter file is over 150 lines.

4 Discussion

To make the transition from pure description to an executable definition language, ESDL requires an execution engine and a collection of domain operators. An execution engine runs the ESDL system and may be an interpreter, a compiler or some hybrid of the two. ESDL could also be transformed into configuration formats used by other frameworks, such as ECJ’s parameter lists (shown in Listing 11), provided the underlying system is capable of supporting the algorithm. We envisage a future where a flexible interpreted system is used to design an algorithm in ESDL, which is then transferred, without modification, to a compiler for executing the algorithm on a supercomputer, a parameter-tuning framework or directly into a business application.

A collection of basic selection, joining and variation operators is required, as is the ability to specify custom operators. There is no standard set of operators for ESDL as yet, and while defining a standard library of common operators would be a significant undertaking that is ultimately beneficial to the entire EC field, it is beyond the scope of this paper.

The lack of conditional statements, such as if and while, is deliberate, since limiting the number of paths through a system significantly reduces the complexity. Systems described using ESDL are effectively fixed data-flow models that define how individuals move between various groups over time. This omission does not limit the ability of ESDL to describe adaptive systems, as complex operations can and should be specified separately in a more appropriate style or language. The use of multiple blocks as described in [9] allows systems to dynamically change the algorithm’s behaviour beyond that permitted by numeric parameter adjustment.

The esec framework uses ESDL as its configuration format. ESDL is transformed into Python code and executed using the Python interpreter, along with any custom operators or external functions provided. A range of classic EAs are included with the framework, and esec has been used to implement algorithms such as Ant System [10], Differential Evolution [12] and Particle Swarm

---

Listing 11: Koza’s GP system [19] partially expressed using ECJ’s parameter file format

```plaintext
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.numbtrees = 1
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
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 12: Koza’s GP system [19] expressed using ESDL

```plaintext
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
```

---

11Based on Java’s property list format, which uses a key=value notation.
12The complete description is available at http://cs.gmu.edu/~eclab/projects/ecj/docs/parameters.html
13Online at http://esec.googlecode.com
Optimisation [13], producing results comparable to existing work. The esecui project\footnote{Online at \url{http://esecui.googlecode.com/}} provides a graphical integrated development environment for designing and testing ESDL-based algorithms.

5 Summary

This work has shown that a common language for describing algorithms in evolutionary computation could be beneficial to reproducibility. The present state of algorithm descriptions limits independent verification and potentially prevents valuable 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 abstracted from the behaviour and implementation of domain operators. With examples we have demonstrated that ESDL is capable of representing a range of evolutionary algorithms unambiguously. 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 parameters for existing frameworks, while most other forms of description require manual interpretation. By removing or simplifying this translation process, algorithms expressed in ESDL are easier to use correctly and are more likely to be used by the wider evolutionary computation community.

Work is already progressing on a clear and robust specification of the language and conceptual model, which will simplify the task of adding ESDL support to other frameworks besides esec. The creation of a standard library of operators has also been highlighted as essential to sharing algorithms amongst researchers. Finally, an evaluation of the readability and adequacy of ESDL is planned, with the results to be used to direct improvements and changes to the language.

References


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


