Specifying Particle Swarm Optimisation in ESDL

Steve Dower* Clinton Woodward

Swinburne University of Technology
Melbourne, Australia
November 10, 2010

Abstract

Evolutionary Systems Definition Language (ESDL) is a domain-specific language for search algorithms based on iterative improvements to a solution population. Particle Swarm Optimisation (PSO) is a population-based search algorithm based on models of swarm behaviour. This report describes PSO using ESDL and validates the performance against earlier published PSO work.

1 Introduction

1.1 ESDL

Evolutionary System Definition Language (ESDL) is a domain-specific language for describing the flow of evolutionary algorithms (EAs) [1]. It defines a system as *groups* of *individuals* and the process followed to create new groups from existing ones. Groups are created by selecting and modifying individuals from existing groups or from *generators*.

ESDL does not specify the implementation or behaviour of specific operations: it requires an underlying framework to provide the functionality. For this report, `esec`[1] is the supporting framework.

1.2 Particle Swarm Optimisation

Particle Swarm Optimisation (PSO) is a search algorithm introduced by Kennedy and Eberhart [2] that uses a population of individuals "flying" within the solution space. The velocity of each individual is determined by the best location found by itself and the best location found by any member of its neighbourhood. Velocity and position are updated each generation, taking the place of a traditional EA crossover and mutation scheme.

The velocity calculation is

$$v_{id} = w \cdot v_{id} + c_1 r_1 (p_{id} - x_{id}) + c_2 r_2 (p_{gd} - x_{id}) \tag{1}$$

where x_{id} and v_{id} are the position and velocity of particle i in dimension d, p_{id} is the best position seen by particle i, p_{gd} is the best position seen by any neighbour, w is an inertial weight, c_1 and c_2 are positive acceleration constants and r_1 and r_2 are random numbers in the range $[0,\ 1]$ taken from a uniform distribution each time the equation is evaluated.

The position update is

$$x_{id} = x_{id} + v_{id} \tag{2}$$

where x_{id} and v_{id} are defined for (1).

*Contact via http://stevedower.id.au/

[1] Available online at http://code.google.com/p/esec. The plugin and configuration files used in this report are also available from here.

Velocity values are typically limited to the same range as valid position values to ensure that even at the maximum velocity, a particle is unable to move over the entire area in a single step. Clerc proposed the use of constriction factors as an alternative to velocity clamping [3, 4]. The resultant velocity in (1) is multiplied by a constant

$$K = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|} \tag{3}$$

where $\varphi = c_1 + c_2$ and is greater than four. Alternatively, K may be multiplied through w, c_1 and c_2 of (1) to provide adjusted values for these parameters.

2 System Definition

ESDL can be used to define PSO applied to an n-dimensional function as shown in Listing 1. In this definition, particles use a global neighbourhood of every other particle. The externally defined elements are `random_pso`, `update_velocity` and `update_position`. All other elements are provided by `esec`; descriptions of their behaviour are included below for completeness. The problem landscape is specified externally. Changing the problem landscape may require modification of the numeric values in line 1.

Listing 1: ESDL definition of the PSO algorithm

```
1  FROM random_pso(length=10, lowest=-100, highest=100) \
2          SELECT (size) population
3
4  FROM population SELECT 1 global_best USING best
5  FROM population SELECT (size) p_bests
6  YIELD population
7
8  BEGIN GENERATION
9      JOIN population, p_bests INTO pairs USING tuples
10
11     FROM pairs SELECT population USING \
12         update_velocity(global_best=global_best, w=1.0, c1=2.0, c2=2.0), \
13         update_position
14
15     JOIN population, p_bests INTO pairs USING tuples
16     FROM pairs SELECT p_bests USING best_of_tuple
17
18     FROM population, global_best SELECT 1 global_best USING best
19
20     YIELD global_best, population
21 END GENERATION
```

The `random_pso` generator produces individuals with uniformly distributed positions and velocities within the provided bounds. The global best is initially determined on line 4 and updated each generation on line 18 by selecting the single best individual from the current population and the previous best. Each individual's personal best is determined by selecting the best out of each individual and its previous personal best, performed using the `JOIN`–`INTO` instruction on line 15 and the `best_of_tuple` selector on line 16.

Behaviour for the `update_velocity` operator is specified in Listing 2. It is applied to a tuple containing each particle, its best previous location – created by the `JOIN`–`INTO` instruction on line 9 of Listing 1 – and the current best individual, `global_best`.

Listing 2: Pseudocode for `update_velocity`

```
function update_velocity(source, global_best, w, c1, c2):
    use w = 1.0, c1 = 2.0, and c2 = 2.0 if not provided
    Pg =first (only) individual in global_best

    for each (Xi and Vi) and Pi in source:
        for each xi, vi, pi and pg in Xi, Vi, Pi, and Pg:
            vi = wvi + c1r1 (pi − xi) + c2r2 (pg − xi)

        yield Xi and updated Vi
```

Positions of individuals are updated by `update_position` as specified in Listing 3.

Listing 3: Pseudocode for `update_position`

```
function update_position(source):
    for each Xi and Vi in source:
        for each xi and vi in Xi and Vi:
            xi = xi + vi

        yield new Xi
```

Bounds enforcement on velocity and position vectors is not shown here, but may be easily added to Listing 2 or Listing 3 as appropriate. The PSO plugin for `esec` supports position wrapping, clamping and bouncing and velocity clamping if desired.

3 Validation

While the ESDL description has well-defined behaviour, an empirical analysis is used to verify that this behaviour matches existing PSO publications.

As a benchmark, we used the methodology and results published in [5]. Five problem landscapes were selected: Sphere, Rosenbrock, Rastrigin, Griewangk and Schaffer's f6. (Details are provided in Appendix A.) Each problem was run with two parameter sets and three population sizes, resulting in 30 configurations.

The first parameter set, "A", was found using a combination of theoretical analysis and experimentation [5]. Set "A" uses $w = 0.6$ and $c_1 = c_2 = 1.7$ as values to equation (1). Parameter set "B" uses equation (3) with $c_1 = c_2 = 2.05$ to find $K = 0.7298$. Distributing K through (1) gives $w = 0.7298$ and $c_1 = c_2 = 1.496$, which are different to the values used in [5]. For consistency with [5], we used $w = 0.7290$ and $c_1 = c_2 = 1.494$.

Each problem and parameter set was run with 15, 30 and 60 individuals in the swarm. Positions and velocities were randomly initialised within the maximum position range for each problem and neither position nor velocity was constrained during the run. Each configuration was run 20 times and the average (mean) number of generations recorded. Generations were capped at 10 000 after which the run was deemed unsuccessful and terminated. Unsuccessful searches were not included in averages.

Average generation count and success rate for the full set of configurations from [5] and the ESDL implementation are shown in Table 1. Averages are shown to four significant figures and success rates as shown as a fraction, zero indicating no successes and one indicating that every run succeeded.

Table 1: Comparison of each configuration

			Mean Generations		Success Rate	
Problem	**Set**	**Pop.**	**Benchmark**	**ESDL**	**Benchmark**	**ESDL**
Sphere	A	15	769.0	802.0	0.40	0.15
		30	344.0	323.7	1.00	1.00
		60	252.0	258.1	1.00	1.00
	B	15	764.0	721.8	1.00	1.00
		30	395.0	396.1	1.00	1.00
		60	314.0	294.0	1.00	1.00
Rosenbrock	A	15	531.0	695.3	0.50	0.15
		30	614.0	1004	1.00	1.00
		60	337.0	250.1	1.00	0.95
	B	15	1430	1332	1.00	1.00
		30	900.0	467.4	1.00	1.00
		60	611.0	375.7	1.00	1.00
Rastrigin	A	15	172.0	229.6	0.35	0.70
		30	140.0	129.3	0.90	0.80
		60	122.0	105.4	0.95	0.95
	B	15	299.0	245.8	0.80	0.90
		30	182.0	155.2	0.95	0.90
		60	166.0	152.3	1.00	0.95
Griewangk	A	15	689.0	526.3	0.35	0.45
		30	313.0	282.8	0.90	0.95
		60	266.0	224.1	0.95	0.95
	B	15	755.0	553.3	0.60	0.55
		30	365.0	363.7	0.90	1.00
		60	287.0	264.9	1.00	1.00
Schaffer's f6	A	15	583.0	568.5	0.45	0.40
		30	161.0	195.0	0.75	0.70
		60	169.0	301.2	0.90	0.85
	B	15	1203	1478	0.40	0.30
		30	350.0	438.0	0.60	0.70
		60	319.0	283.0	0.95	0.95

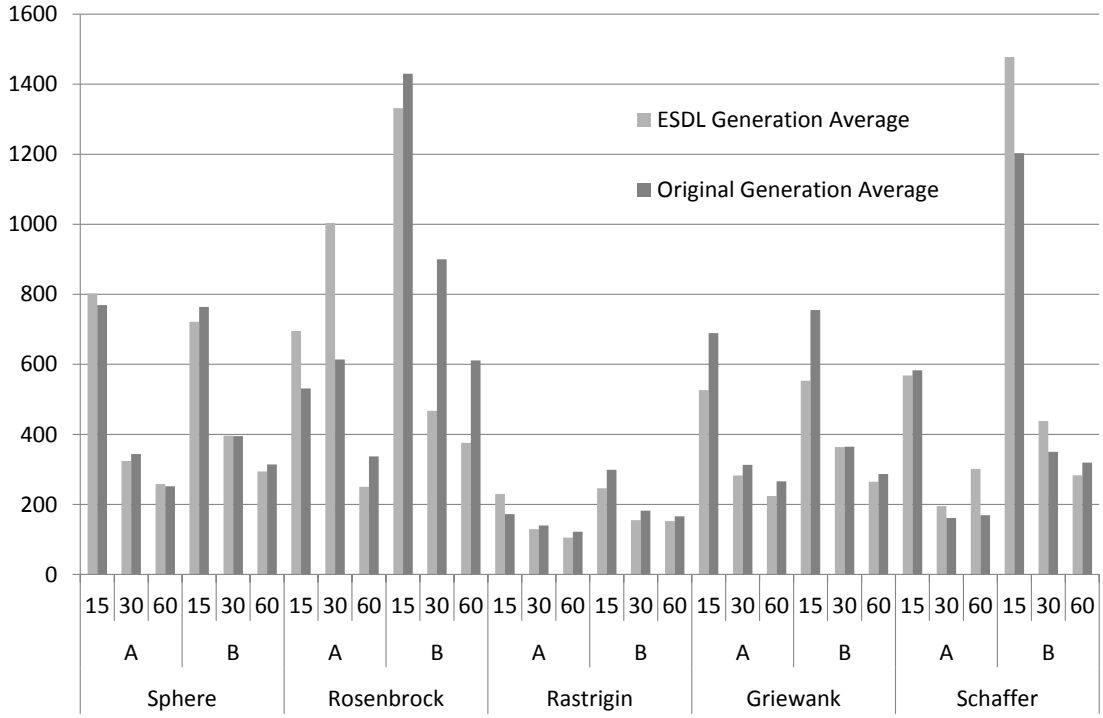

Figure 1: Average generation count for each configuration

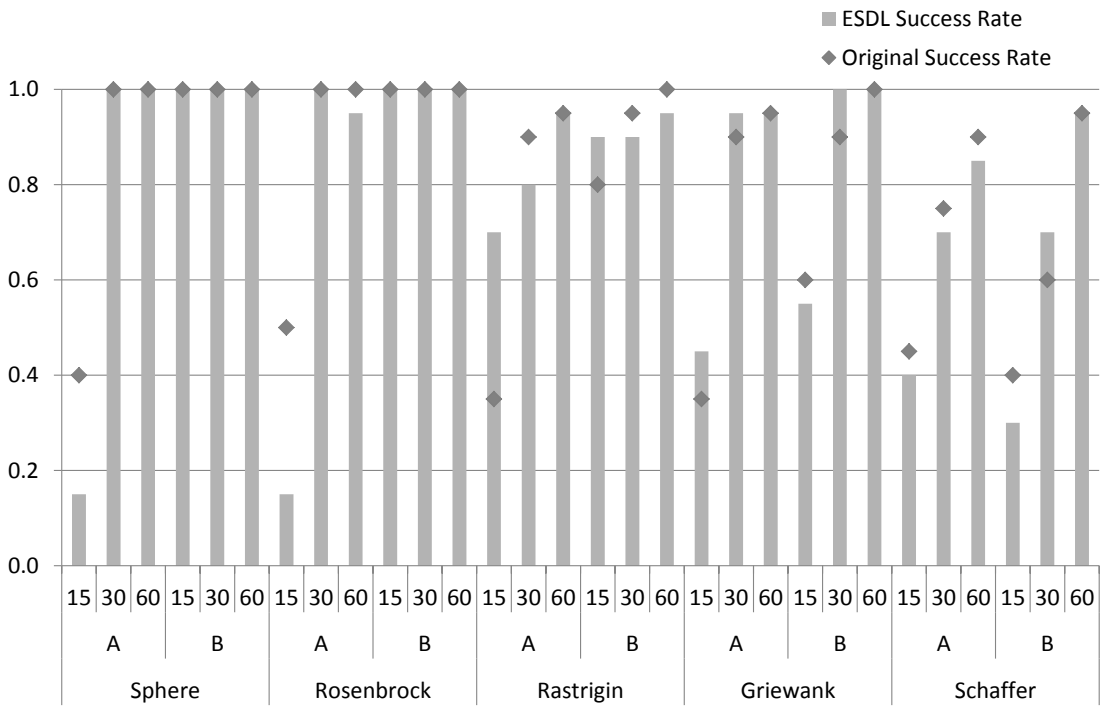

Figure 2: Success rate for each configuration

Figures 1 and 2 show that the ESDL-based implementation produces similar results to those published in [5]. In those cases the performance observed in the ESDL implementation is significantly different (such as for Rosenbrock), the small number of experiments conducted for each configuration is suspected to be the cause. As a number of repeats of these configurations produced significantly different results, no particular set being any more valid, we present one set of results here and recommend the use of a larger sample for proper statistical analysis.

Another observation of the validation data is that small populations with parameter set "A" exhibited low success rates. With one exception, each run with a population of 15 had less than a fifty per cent likelihood of finding a solution. Since unsuccessful experiments were not included in each of the mean generation counts, the averages tend to be biased lower, which is obvious in the results for Rosenbrock against parameter set "A". This is an artefact of the algorithm and has no bearing on the validity of our results.

From the results presented, it can be clearly demonstrated that the ESDL implementation matches the algorithm described in [5].

4 Summary

This report provides a description of the Particle Swarm Optimisation algorithm using ESDL. The fundamental algorithm is shown concisely in ESDL with the velocity and position update methods specified using pseudocode.

The ESDL description was used with `esec` to produce a working implementation, which was validated against five n-dimensional problem landscapes. Results show that the ESDL implementation is consistent with existing published results.

References

[1] S. Dower and C. Woodward, "Evolutionary System Definition Language," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/1, 2010. 1

[2] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4. IEEE, 1995. 1

[3] M. Clerc, "The swarm and the queen: Towards a deterministic and adapative particle swarm optimization," in *Evolutionary Computation, 1999., IEEE International Conference on.* IEEE, 1999. 2

[4] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, February 2002. 2

[5] I. C. Trelea, "The particle swarm optimization algorithm: convergence analysis and parameter selection," *Information Processing Letters*, vol. 85, 2003. 3, 6, 7

[6] H. H. Rosenbrock, "An automatic method for finding the greatest or least value of a function," *The Computer Journal*, vol. 3, 1960. 7

A Test Functions

All of these functions are minimisation problems with the global minimum $f_{min} = 0$ at $x_* = 0$. The value of f_{goal} is the largest fitness value that represents a successful run.

The Sphere function is a simple sum of each parameter squared:

$$f(x) = \sum_{i=1}^{n} x_i^2 \tag{4}$$

In this report, $n = 30$, $x \in [-100,\ 100]$ and $f_{goal} = 0.01$.

Rosenbrock's valley is a well-known classic optimisation problem [6] with many alternative titles, such as De Jong's Function 2 (F2), Rosenbrock's "saddle" and the "Banana" function. The function is continuous and unimodal, though the non-convex surface gradient can be deceptive to some search methods. Originally a two-dimensional function, it has a variety of generalisations to n-dimensions. The version used in this report is:

$$f(x) = \sum_{i=1}^{n-1} \left(100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right) \tag{5}$$

In this report, $n = 30$, $x \in [-30,\ 30]$ and $f_{goal} = 100$.

Rastrigin is a multimodal and deceptive surface, similar to the Sphere but with an added cosine term to create multiple local minima:

$$f(x) = \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right) \tag{6}$$

In this report, $n = 30$, $x \in [-5.12,\ 5.12]$ and $f_{goal} = 100$.

Griewangk (sometimes referred to as Griewank) is based on adding a modulation term to the Sphere function. At a large scale, the surface appears smooth, while at small scales close to the optimum it becomes very deceptive.

$$f(x) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{7}$$

In this report, $n = 30$, $x \in [-600,\ 600]$ and $f_{goal} = 0.1$.

Schaffer's f6 function is a two-dimensional landscape that is primarily flat, but becomes deceptive close to the global minimum:

$$f(x) = 0.5 + \frac{\sin^2\sqrt{x_1^2 + x_2^2} - 0.5}{\left(1 + 0.001\left(x_1^2 + x_2^2\right)\right)^2} \tag{8}$$

In this report, $n = 2$, $x \in [-100,\ 100]$ and $f_{goal} = 10^{-5}$. (The version of this equation given in [5] appears to be an inverted form, with a global maximum $f_{max} = 1$ at $x_* = 0$. However, the other parameters and results suggest that the minimisation landscape was used.)

B esec Configuration

Listing 4 shows the configuration used for the Sphere experiment with parameter set A and a population of 30. The other configurations were slight variations of the one shown; the ESDL definition was not changed.

Listings 5 and 6 show the implementation of an individual and species for PSO, which allow better integration with `esec` and simpler reuse. The `update_velocity` and `update_position` methods shown in listings 7 and 8 respectively are based on the pseudocode in listings 2 and 3.

(The code shown has been simplified and comments removed from the full code available online at http://code.google.com/p/esec/.)

Listing 4: The `esec` configuration used for the Sphere landscape

```
from esec.landscape import real
import plugins.PSO

config = {
    'landscape': {
        'class': real.Sphere,
        'size': { 'exact': 30 }
    },
    'system': {
        'definition': r'''
FROM random_pso(length=(n), lowest=(-x_max), highest=(x_max)) \
                SELECT (size) population
FROM population SELECT 1 global_best USING best_only
FROM population SELECT (size) p_bests
YIELD population

BEGIN GENERATION
    JOIN population, p_bests INTO pairs USING tuples

    FROM pairs SELECT population USING \
         update_velocity(global_best=global_best, w=w, c1=c1, c2=c2), \
         update_position

    JOIN population, p_bests INTO pairs USING tuples
    FROM pairs SELECT p_bests USING best_of_tuple

    FROM population, global_best SELECT 1 global_best USING best_only

    YIELD global_best, population
END GENERATION''',
        'size': 30,
        'n': 30, 'x_max': 100,
        'w': 0.6, 'c1': 1.7, 'c2': 1.7
    },
    'monitor': {
        'report': 'brief+local+time_delta',
        'summary': 'status+brief+best_genome',
        'limits': {
            'generations': 10000,
            'fitness': 0.01,
        }
    },
}
```

Listing 5: Python implementation of the `PSOIndividual` class

```
class PSOIndividual(RealIndividual):
    # Actually a real-valued individual, but we'll store velocity values
    # in the vector as well. The phenome and velocities properties can
    # sort it out.

    @property
    def genome_string(self):
        '''Returns a string representation of the genes of this individual.'''
        return '[' + ', '.join('%g (%+g)' % i \
            for i in zip(self.phenome, self.velocities)) + ']'

    @property
    def phenome(self):
        '''Returns the position values for this individual.'''
        return self.genome[:len(self.genome)//2]

    @property
    def position_bounds(self):
        '''Returns the bounds for positions for this individual.'''
        i = len(self.genome) // 2
        return (self.bounds[0][:i], self.bounds[1][:i])

    @property
    def velocities(self):
        '''Returns the velocity values for this individual.'''
        return self.genome[len(self.genome)//2:]

    @property
    def velocity_bounds(self):
        '''Returns the bounds for velocities for this individual.'''
        i = len(self.genome) // 2
        return (self.bounds[0][i:], self.bounds[1][i:])

    @property
    def phenome_string(self):
        '''Returns a string representation of the phenome of this
        individual.'''
        return '[' + ', '.join('%.3f' % p for p in self.phenome) + ']'
```

Listing 6: Python implementation of the `PSOSpecies` class and `init_random`

```
class PSOSpecies(RealSpecies):
    name = 'pso'

    def __init__(self, cfg, eval_default):
        super(PSOSpecies, self).__init__(cfg, eval_default)
        # Make some names public within the ESDL definition
        self.public_context = {
            'random_pso': self.init_random,
            'update_velocity': update_velocity,
            'update_position': update_position,
        }

    def init_random(self, length=2, lowest=-1.0, highest=1.0, \
                    zero_velocity=False, template=None):
        frand = rand.random

        # Specify bounds, but don't necessarily apply them later
        bounds = ([lowest]*length, [highest]*length)

        if zero_velocity:
            # Initialise velocity to zero
            for indiv in self._init(length, None, None, \
                                    lowest, highest, bounds, template, \
                                    lambda l, h, _: frand()*(h-l)+l):
                yield PSOIndividual(indiv.genome+[0]*length, self, indiv.bounds)
        else:
            # Initialise velocity to within the same range as the
            # position.
            for indiv in self._init(length * 2, None, None, \
                                    lowest, highest, bounds, template,
                                    lambda l, h, _: frand()*(h-l)+l):
                yield PSOIndividual(indiv.genome, self, indiv.bounds)
```

Listing 7: Python implementation of `update_velocity` (based on Listing 2)

```
def update_velocity(source, global_best, w=1.0, c1=2.0, c2=2.0):
    global_best = global_best[0]

    frand = rand.random

    for joined_individual in source:
        indiv, indiv_best = joined_individual[:]

        new_velocity = list(indiv.velocities)
        for i, (pos, vel, pbest_pos, gbest_pos) in \
            enumerate(zip(indiv, new_velocity, indiv_best, global_best)):

            new_vel = w*vel + c1*frand()*(pbest_pos-pos) + \
                              c2*frand()*(gbest_pos-pos)
            new_velocity[i] = new_vel

        yield PSOIndividual(indiv.phenome + new_velocity, indiv)
```

Listing 8: Python implementation of `update_position` (based on Listing 3)

```
def update_position(source):
    for indiv in source:
        new_position = list(indiv)
        velocity = list(indiv.velocities)

        for i, (pos, vel) in enumerate(zip(new_position, velocity)):
            new_position[i] = pos + vel

        yield PSOIndividual(new_position + velocity, indiv)
```