Towards the universal framework of stochastic nature-inspired population-based algorithms

Iztok Fister Jr., Janez Brest, Uroš Mlakar, Iztok Fister
Faculty of Electrical Engineering and Computer Science
University of Maribor
Smetanova 17, 2000 Maribor
Slovenia
Email: iztok.fister1@um.si

Abstract—**Stochastic nature-inspired population-based algorithms have attracted a huge community of researchers and practioners for who use them for optimization purposes. The implications of using the methods are almost unlimited. Most of these nature-inspired algorithms are inspired by the biological principles of behavior of various animals living in nature. In our opinion, a lot of research has turned in the wrong direction recently. Instead of improving existing algorithms, they have been proposing the new algorithms discontinuously. Unfortunately, most of these so called "new nature-inspired algorithms" are actually modifications of already known algorithms hidden behind the pompous metaphor-based inspiration. In this theoretically oriented paper, we propose a universal framework of stochastic nature-inspired population-based algorithms that try to unify the more real-coded evolutionary and swarm intelligence based algorithms under the same umbrella. In line with this, a domain-specific embedded language is proposed that enables a generation of the more important stochastic real-coded nature-inspired population-based algorithms on the one hand, and creation of the new ones on the other hand. Consequently, this approach proves that the gap between the nature-inspired algorithms is actually not so big as seems at the first sight.**

I. Introduction

Observing the nature, tracing the ant trails, watching how termites build their nests (also mounds), inspecting wolves and their hunting habits in the deep forests, studying the flying traces of birds, and even watching small lighting bugs called fireflies in the young summer nights, had led to extensive development of computer algorithms based on nature. Actually, all these behaviors of animals and insects can be observed as an optimization process. In line with this, a mathematical formulation for describing these processes can be used within these algorithms. As a result, such algorithms can be applied for solving the hardest optimization problems.

Two families of algorithms have been arisen on this basis, i.e., Evolutionary Algorithms (EAs) and Swarm Intelligence (SI) based algorithms. The former have been inspired by the Darwinian struggle for existence [3], while the latter by mimicking the behavior of small creatures (also agents) that together in a community are capable of performing great jobs, like the already mentioned ants, termites and fireflies [8].

Recently, the development of the new stochastic nature-inspired population-based algorithm was misused slightly. We are witnesses of the almost every month occurrence of the new nature-inspired algorithms that, unfortunately, do not bring anything new, but try to convince their potential users with pompous metaphor [15]. Actually, almost each process in the nature can be described as an optimization process. Therefore, the potential inspirations for the new nature-inspired algorithms are countless.

The purpose of this paper is to propose an universal framework which could enable development of these new algorithms automatically. In line with this, the existing real-coded EAs and some SI-based algorithms are analyzed. The common characteristics of these algorithms are then represented in a feature diagram that is used for developing the Domain-Specific Embedded Language (DSEL) [9] in Ruby. The specifications of this DSEL inherit the generic language constructs for the Ruby interpreter and add domain-specific primitives for creating the specific stochastic nature-inspired population-based algorithms. At this moment, this interpreter is capable of generating the Differential Evolution (DE) [16], [4] an algorithm that belongs to the EAs family as well as the more of the real-coded SI-based algorithms, like Particle Swarm Optimization (PSO) [10], Firefly Algorithm (FA) [6], Cuckoo Search (CS) [19], [11] and Bat Algorithm (BA) [18].

The paper is organized as follows. In Section II, an analysis is performed of the stochastic nature-inspired population-based algorithms. Section III describes the development of the universal framework for developing the stochastic nature-inspired population-based algorithms. The preliminary experiments and results are the subjects of Section IV. In Section V, the paper summarizes the performed work and outlines directions for the future work.

II. Analysis of nature-inspired population-based algorithms

In this section the characteristics of the EAs, like DE, and SI-based algorithms, like PSO, FA, CS and BA, are analyzed in order to show that these algorithms actually have very similar internal structures. Identifying these common characteristics can serve as a basis for development of the universal framework for generating the stochastic nature-inspired population-based algorithms in the future. The section is divided into two subsections. The former is devoted to description of the DE algorithm, and the latter for identifying the common SI-based framework.

A. Differential evolution

Differential Evolution (DE) [4] is an evolutionary algorithm appropriate for continuous and combinatorial optimization that was introduced by Storn and Price [16] in 1995. This is a population-based algorithm that consists of Np real-coded vectors representing the candidate solutions, as follows:

$$\mathbf{x}_i^{(t)} = (x_{i,1}^{(t)}, \ldots, x_{i,D}^{(t)}), \quad \text{for } i = 1, \ldots, Np. \tag{1}$$

The variation operator in DE supports a differential mutation and a differential crossover. In particular, the differential mutation selects two solutions randomly and adds a scaled difference between these to the third solution. This mutation can be expressed as follows:

$$\mathbf{u}_i^{(t)} = \mathbf{x}_{r1}^{(t)} + F \cdot (\mathbf{x}_{r2}^{(t)} - \mathbf{x}_{r3}^{(t)}), \quad \text{for } i = 1, \ldots, Np, \tag{2}$$

where F denotes the scaling factor as a positive real number that scales the rate of modification while $r1$, $r2$, $r3$ are randomly selected values in the interval $1 \ldots Np$. Note that typically the interval $F \in [0.1, 1.0]$ is used in the DE community.

As a differential crossover, uniform crossover is employed by the DE, where the trial vector is built from parameter values copied from two different solutions. Mathematically, this crossover can be expressed as follows:

$$w_{i,j}^{(t)} = \begin{cases} u_{i,j}^{(t)} & \text{rand}_j(0,1) \leq CR \vee j = j_{rand}, \\ x_{i,j}^{(t)} & \text{otherwise,} \end{cases} \tag{3}$$

where $CR \in [0.0, 1.0]$ controls the fraction of parameters that are copied to the trial solution. Note, the relation $j = j_{rand}$ ensures that the trial vector is different from the original solution $\mathbf{x}_i^{(t)}$.

A differential selection is, in fact, a generalized one-to-one selection that can be expressed mathematically as follows:

$$\mathbf{x}_i^{(t+1)} = \begin{cases} \mathbf{w}_i^{(t)} & \text{if } f(\mathbf{w}_i^{(t)}) \leq f(\mathbf{x}_i^{(t)}), \\ \mathbf{x}_i^{(t)} & \text{otherwise}. \end{cases} \tag{4}$$

In a technical sense, crossover and mutation can be performed in several ways in differential evolution. Therefore, a specific notation is used to describe the varieties of these methods (also strategies) generally. For example, 'DE/rand/1/bin' denotes that the base vector is selected randomly, one vector difference is added to it, and the number of modified parameters in the mutant vector follows binomial distribution. A detailed description of the other DE mutation strategies, as well as exponential crossover, can be seen in [4], [12].

The pseudo-code of a DE algorithm is presented in Algorithm 1, from which it can be seen that DE consists of the following components:

- initialization (line 1),
- fitness function evaluation (lines 2 and 5),
- termination condition (line 3),
- variation operators (line 4),
- replacement operator (line 6),
- finding the best solution (line 7).

Algorithm 1 The original DE algorithm

```
1: INITIALIZE_population_randomly;
2: EVALUATE_each_candidate_solution;
3: while TERMINATION_CONDITION_not_met do
4:     MODIFY_candidate_solutions_using_DE_mutation_strategy;
5:     EVALUATE_each_trial_solution;
6:     REPLACE_candidate_solutions;
7:     FIND_global_best_solution;
8: end while
```

However, the representation of solution is a prerequisite for the algorithm to work correctly.

B. SI-based framework

Pseudo-codes of the following algorithms were analyzed in order to identify which characteristics are common to the majority of the SI-based algorithms: PSO, FA, CS and BA. The results of the analysis are presented in Table I from which it can be seen that the typical SI-based algorithm consists of five mandatory components and one or more optional.

The mandatory components in Table I are as follows:

- initialization of an initial population randomly (INITIALIZE),
- fitness function evaluation (EVALUATE),
- modification operator (MODIFY),
- replacement operator (REPLACE),
- finding the global solution (FIND) and
- optional restart operator (RESTART).

All the considered algorithms are population-based, where the population consists of Np population members representing solution instances of the problem, i.e., vectors $\mathbf{x}_i = (x_{i,0}, \ldots, x_{i,D})$ with real-valued elements of dimension D. The population members are called particles, in this paper.

Mostly, the population of particles is initialized randomly according to the equation:

$$x_{i,j}^{(0)} = (Ub_j - Lb_j) \cdot \mathrm{U}(0,1) + Lb_j, \tag{5}$$

where $\mathrm{U}(0,1)$ denotes a randomly generated number drawn from uniform distribution in an interval $[0,1]$, while Ub_j and Lb_j are the upper and lower bounds of the generated element, respectively.

Although all these algorithms are very similar, they differ between each other in the way they modify a candidate solution from the current position to a new, possibly better position. The modify operation in SI-based algorithms is usually inspired by the behavior of the various natural systems and it prescribes the way in which the particles are moved in the search space.

The PSO algorithm mimics the behavior of a shoal of fish, a swarm of insects or a herd of animals, where the new particle position $\mathbf{x}_i^{(t+1)}$ depends on the velocity $\mathbf{v}_i^{(t+1)}$. This velocity is obtained by its current velocity $\mathbf{v}_i^{(t)}$ affected by its best

TABLE I
CHARACTERISTICS OF THE OBSERVED SI-BASED ALGORITHMS

Component	PSO	FA	CS	BA
INITIALIZE	random	random	random	random
EVALUATE	problem dependent	problem dependent	problem dependent	problem dependent
MODIFY	PSO mutation strategy	FA mutation strategy	CS mutation strategy	BA mutation strategy
REPLACE	extra-memory	extra-memory	one-to-rand	one-to-one-cond
FIND	deterministic	deterministic	deterministic	deterministic
RESTART	n/a	n/a	optional	n/a

position $\mathbf{p}_i^{(t)}$ (i.e., the personal best) as well as the position of the best particle $\mathbf{g}_i^{(t)}$ (i.e., the global best) in the swarm. As a result, the new particle position is expressed, as follows:

$$\begin{aligned}\mathbf{v}_i^{(t+1)} &= \mathbf{v}_i^{(t)} + F(\mathbf{p}_i^{(t)} - \mathbf{x}_i^{(t)}) + K(\mathbf{g}_i^{(t)} - \mathbf{x}_i^{(t)}),\\ \mathbf{x}_i^{(t+1)} &= \mathbf{x}_i^{(t)} + \mathbf{v}_i^{(t+1)},\end{aligned} \tag{6}$$

where $F = C_1\mathcal{U}(0,1)$ and $K = C_2\mathcal{U}(0,1)$ are scale factors. Thus C_1 and C_2 denote the learning factors.

The FA algorithm mimics the flashing behavior of fireflies that can be expressed mathematically as:

$$\mathbf{x}_i^{(t+1)} = \begin{cases} \mathbf{x}_i^{(t)} + \beta(\mathbf{x}_j^{(t)} - \mathbf{x}_i^{(t)}) + \alpha^{(t)}\epsilon^{(t)}, & \text{if } f(\mathbf{x}_j) \leq f(\mathbf{x}_i) \\ \mathbf{x}_i^{(t)}, & \text{otherwise,} \end{cases} \tag{7}$$

where $\beta = \beta_0 \exp -\gamma r_{i,j}^2$ is a scaling factor and $\alpha^{(t)}$ the randomization parameter.

The CS algorithm is inspired by the brood parasitism of some cuckoo species that is expressed mathematically as

$$\mathbf{y}^{(t+1)} = \mathbf{x}_i^{(t)} + \alpha \mathrm{L}(s, \lambda), \tag{8}$$

where α is a scale factor and $\mathrm{L}(s, \lambda)$ represents a random number drawn from the Lévy flights distribution that is explained in next section.

The BA algorithm incorporates an explicit regulation between exploration and exploitation in the search process. Thus, the exploitation component of the search process is expressed as

$$\mathbf{y}^{(t+1)} = \mathbf{x}_{best}^{(t)} + \alpha \cdot \mathbf{u}^{(t)}, \tag{9}$$

where α denotes a scale factor (step length) and $\mathbf{u}^{(t)}$ is a random vector of elements drawn from the uniform distribution in the interval $[0,1]$. In fact, this equation represents a random walk-based direct exploitation method [13] and tries to find a better solution in the vicinity of the current best. The global search (exploration) exploits a phenomenon of echolocation that is expressed in the form of a move operator as follows

$$\mathbf{y}^{(t+1)} = \mathbf{x}_i^{(t)} + F_i^{(t)} \cdot (\mathbf{x}_i^{(t)} - x_{best}^{(t)}), \tag{10}$$

where $F_i^{(t)}$ denotes a scaling factor and $\mathbf{x}_{best}^{(t)}$ is the best solution found until now.

In general, a pseudo-code of the common SI-based framework is obtained as illustrated in Algorithm 2. At a glance, this pseudocode resembles the pseudocode illustrated in Algorithm 1. Actually, the only difference between both pseudocodes lies in a presence of component RESTART (line 8 in Algorithm 2). Indeed, this component is even optional.

Algorithm 2 A pseudo-code of the SI-based algorithm

```
1: INITIALIZE_population_with_random_generated_particles;
2: EVALUATE_each_particle;
3: while TERMINATION_CONDITION_not_met do
4:     MODIFY_particles_using_specific_mutation_strategy;
5:     EVALUATE_trial_particles;
6:     REPLACE_particles;
7:     FIND_best_particle_for_the_next_generation;
8:     RESTART_the_worst_particle;
9: end while
```

III. DESIGN OF THE UNIVERSAL FRAMEWORK

Development of the proposed framework starts with a domain analysis, where a feature model is obtained. Based on the feature model, the semantics of the Domain-Specific Embedded Language (DSEL) for the Ruby host language is defined that have the "look and feel" of the syntax. Then, this DSEL specification is translated to the source code of the desired stochastic nature-inspired population-based algorithm.

In a nutshell, the proposed approach consists of the following three steps:

- domain analysis,
- domain specific semantics,
- source code interpretation.

In the remainder of the paper, all steps are described in detail.

A. Domain analysis

The purpose of domain analysis is to identify the characteristics of the observed EA and SI-based algorithms. These characteristics are usually presented in the form of Feature Diagrams (FD) [7], [14], [17] by designing the domain-specific languages. Thus, the characteristics of the typical stochastic nature-inspired population-based algorithms are identified and represented as a FD that serves as a basis for generating the source code of the desired stochastic nature-inspired population-based algorithm by the DSEL specifications.

The FD illustrated in Fig. 1 represents a concept *algorithm* that consists of nodes denoting the features and arcs defining the relationships between nodes. The features can be mandatory or optional as denoted by closed or opened dots ending the arcs, respectively.

In our case, the concept *algorithm* consists of six mandatory features, like initialization (*Init*), representation (*Repr*), fitness function evaluation (*Eval*), variation operators (*Oper*), replacement (*Repl*) and termination condition (*Term*), and one optional feature (*Rest*). These features correspond to components

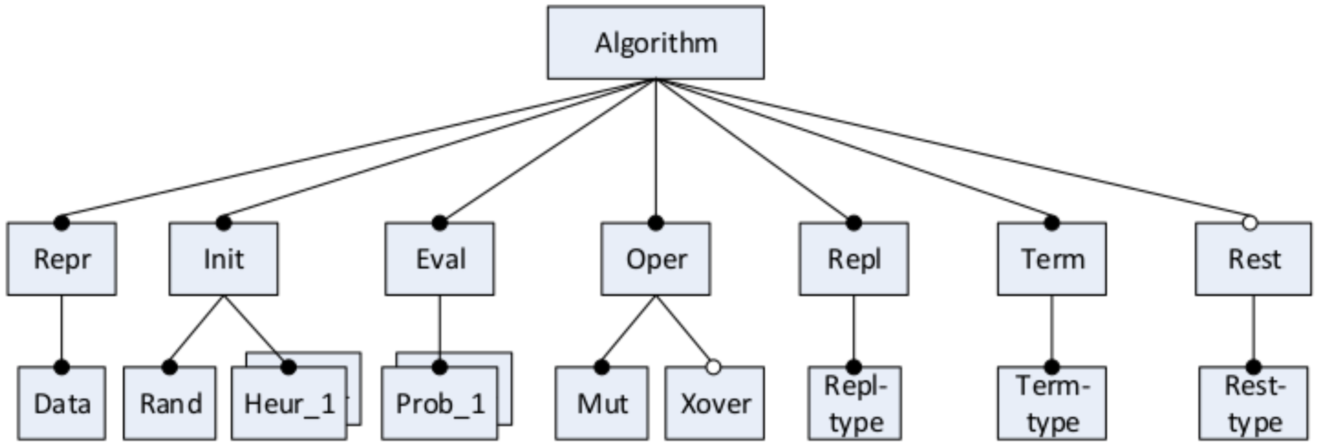

Fig. 1. Feature diagram

of the evolutionary and SI-based algorithms [5]. In our FD, the feature *Repr* consists of defining the variables needed for the proper algorithm execution. The initialization *Init* can be performed either randomly as denoted by sub-feature *Rand* or heuristically as denoted by the sub-features *Heur_1-Heur_*m. The feature *Eval* corresponds to the fitness function evaluation that is problem dependent as denoted by the sub-feature *Prob_1-Prob_*n. The feature *Oper*, corresponding to variation operators, consists of two sub-features, i.e., mutation *Mut* and crossover *Xover*. The former is mandatory, while the latter optional. The feature *Repl* corresponds to the replacement operator and it is specified by various types as determined by the sub-feature *Repl-type*. The feature *Term* corresponding to termination condition consists of one sub-feature *Term-type*. Finally, the optional feature *Rest* corresponds to the restart feature that can also have various types as determined by its sub-feature *Rest-type*.

Algorithm 3 DSEL specification for DE generation in Ruby.

```
 1: DifferentialEvolution = NatureInspiredAlgorithms.build :DE do
 2:   repr(”Data”) {
 3:     D:”20”
 4:     NP:”100”
 5:   }
 6:   init(”Rand”)
 7:   evaluate(”Prob_1”)
 8:   term(”gen”) {
 9:     val:”1000”
10:   }
11:   oper(”Xover”) {
12:     CR:”0.9”
13:   }
14:   oper(”Mut”) {
15:     strategy:”DE/rand/1/bin”
16:     F:”0.5”
17:   }
18:   repl(”one-to-one”)
19: end
```

B. Domain specific semantics

In our study, the DSEL for generating the source code of the specific nature-inspired population-based algorithms was developed in the Ruby host language. This host language is flexible and expressive enough for developing the DSELs. As a result, the DSEL developers are free of reflection about syntax and, therefore, can be focused on the semantic issues of the language that is captured into the interpreter of the DSEL language. In fact, the DSEL can be seen as a high-order algebraic structure that has the ”look and feel” of the syntax [9].

An example of the proposed DESL specifications of the DE algorithm using a ’DE/rand/1/bin’ mutation strategy is presented in Algorithm 3.

In fact, the DSEL specification for DE generation determines the parameters for the features from the FD represented in Fig. 1, for instance, the specification code ”repr(”Data”)” (lines 2-5 in Algorithm 3) defines values of two parameters D and NP for the DE algorithm defining the dimension of the problem and the number of individuals in the population, respectively. The specification code ”init(”Rand”)” (line 6) defines the random initialization of the DE algorithm. It is necessary to solve the ”Prob_1” as specified in the line 7. The specifications ”term(”gen”)” in lines 8-10 determine that The generated algorithm will be terminated after 1000 generations. However, the variation operator is defined by two parts of the specification code illustrated in lines 11-13 and lines 14-17. The first part defines the value probability of crossover CR, and the type and the scale parameter F of the corresponding DE mutation strategy. Finally, the replacement one-to-one is specified by the clause in line 18.

The purpose of this study was also to generate some of the SI-based algorithms. The example of DSEL specification of the BA algorithm is illustrated in Algorithm 4, from which it can be seen that the crossover could be omitted from the

DSEL specification.

Algorithm 4 DSEL syntax specification for BA generation in Ruby.

```
BatAlgorithm = NatureInspiredAlgorithms.build :BA do
  repr(”Data”) {
    D:”20”
    NP:”100”
  }
  init(”Rand”)
  evaluate(”Prob_5”)
  term(”gen”) {
    val:”1000”
  }
  oper(”Mut”) {
    strategy:”BA”
    R:”0.5”
    Qmin:”0.0”
    Qmax:”1.0”
    CR:”1.0”
  }
  repl(”one-to-one-cond”) {
    A:”0.5”
  }
end
```

When comparing the DSEL specifications of both algorithms, an emphasize needs to be considered on a different setting of the parameter CR. That means, if the traditional SI-based algorithm is generated, the crossover parameter is set as $CR = 1.0$, where each element of the population member is modified, while the same parameter is normally set as $CR < 1.0$ in the case of a DE algorithm. On the other hand, using this parameter also shows the big power of the proposed framework because, by modifying this parameter, influence of the crossover operator can also be tested on the SI-based algorithms that typically work without this feature.

C. Source code interpretation

Domain Specific Embedded Language (DSEL) is defined as a library for the generic host programing language. This language is an extension of the host language that is devoted to programers for developing their programs on a much higher level of abstraction. The DSEL inherits the generic language constructs of the host language and adds domain-specific primitives implemented typically as library functions [9]. However, the semantics of DSEL specifications are captured by an interpreter.

In this step, the specification of the stochastic nature-inspired population-based algorithms in DSEL are translated into corresponding Ruby host language source code by the interpreter. The purpose of this subsection is to describe how the DSEL specifications are translated into Ruby source code that implements the corresponding stochastic nature-inspired population-based algorithm. The translation consists of two steps:

- generation of the core of stochastic nature-inspired population-based algorithm,
- implementation of the corresponding library functions.

In the first step, the source code of the stochastic nature-inspired population-based algorithm determined by DSEL specifications is generated by an interpreter. The interpreter generates part of the code corresponding to components of the nature-inspired algorithm in a sequence determined by DSEL specification. This means that the generated program starts with the representation part followed by initialization and evaluation. In evolution cycle terminated by termination condition, the sequence of parts are generated as follows. The replacement part of code is generated after modification and evaluation.

Actually, all parts of the source code representing the components of the generated stochastic nature-inspired population-based algorithm are generated as generic function calls controlled by specific parameters. The different library functions are selected according to the values of these parameters. While the core of the generated nature-inspired algorithm is more or less statical for the observed nature-inspired algorithms, activation of different features is achieved by the passing parameters phase. In fact, the interpreter captures the semantics of DSEL in the passing parameters. The semantics domains of parameters passed to generic functions are illustrated in Table II.

As can be seen in Table II, domain D_{Data} determines a set of variables needed for definition of the generated DSEL program. Domain D_{Init} defines types of initialization by the generated DSEL. These types can be either random (*Rand*) or heuristic (*Heur*). If the heuristic initialization is used, the possible ways of initialization are specified by domain D_{Init_Heur} that enables a lot of heuristic initialization programs denoted by $Heur\_Prob_1$ to $Heur\_Prob_m$. The other semantic domains can be interpreted in a similar way.

The translation of the constructs in DSEL specifications to generic function calls by the interpreter are presented in Table III.

The results of the first step of the translation is a stochastic nature-inspired population-based algorithm in Ruby source code implemented by generic functions calls. These generic functions are implemented as wrappers for calling the appropriate library functions according to the passed parameters. Note that library functions are based on [2] and author's Github repository https://github.com/jbrownlee/CleverAlgorithms.

The second step of the DSEL translation is the implementation of the library functions. The generic parts of the source code obtained by translation of DSEL constructs are presented in the remainder of the paper. Then, the implementation of library functions are discussed as generated by translation of DSEL specifications of DE algorithm (Algorithm 3). Note that the wrapper code is straightforward and therefore its detailed description is omitted in the paper.

1) Data part generation: Data part is generated by translation of the ”repr(”Data”)” DESL specification and it is dedicated for initialization of two variables, d and np, that determine the dimension of the problem and the population size, respectively (Algorithm 5). Let us notice that the vari-

TABLE II
SEMANTIC DOMAINS.

$Data = \{D, Np, pop, fit, trial, trialFit, best, bestFit, lbest\}$	
$D_{Init} = \{Rand, Heur\}$	$Heur \in D_{Init-Heur}$
$D_{Init-Heur} = \{Heur\_1, \ldots, Heur\_m\}$	
$D_{Problem} = \{Prob\_1, Prob\_2, \ldots, Prob\_n\}$	
$D_{Strategy} = \{DE\_strat, PSO\_strat, FA\_strat, CS\_strat, BA\_strat\}$	$DE\_strat \in D_{DE\_strat}, \ldots, BA\_strat \in D_{BA\_strat}$
$D_{DE\_strat} = \{DE\_strat\_type, F, CR, r_1, \ldots\}$	$DE\_strat\_type \in D_{DE\_strat\_type}$
$D_{DE\_strat\_type} = \{rand1bin, \ldots\}$	
$\ldots$	$\ldots$
$D_{BA\_strat} = \{BA\_strat\_type, r, Q_{min}, Q_{max}, CR\}$	$BA\_strat\_type \in D_{BA\_strat\_type}$
$D_{BA\_strat\_type} = \{Normal, Levy\}$	
$D_{repl} = \{$*one-to-one,one-to-one-cond,one-to-rand,extra-memory*$\}$	

TABLE III
TRANSLATION OF DSEL CONSTRUCTS.

repr	$d = a, np = b$	$D \in$ Int, $np \in$ Int
init	init($init\_type$)	$init\_type \in D_{Init}$
evaluate	evaluate($problem$)	$problem \in D_{Problem}$
term	term(gen)	$gen \in D_{Term}$
oper	oper(mut_strat, CR)	$mut\_strat \in D_{Strategy}, CR \in \mathcal{R}$
repl	repl($repl\_type$)	$repl\_type \in D_{Repl}$

ables are written in lower case according to the declaration of the Ruby programing language.

Algorithm 5 DSEL data segment specification in Ruby.

```
1: d = 20
2: np = 100
```

Although the D_{Data} semantic domain demands more variables to be allocated dynamically in the generated program, the main characteristic of the Ruby programing language is that the variables can be allocated dynamically the first time that they are needed. Therefore, there is no need to preallocate the variables at this stage.

2) Initialization generation: Translation of the "init("Rand")" DSEL specification is translated into Ruby source code as presented in Algorithm 6.

Algorithm 6 Generation of DE initialization in Ruby.

```
1: search_space = Array.new(d) |i| [-5, +5]
2: pop = Array.new(np) |i| :vector=>init("Rand", search_space)
```

As can be seen from the code in Algorithm 6, the initialization consists of an allocation of a 2-dimensional array that is initialized randomly. The implementation of the random initialization library function is presented in Algorithm 7.

Algorithm 7 Implementation of DE initialization in Ruby.

```
1: def init_rand(search_space)
2:   return Array.new(search_space.size) do |i|
3:     search_space[i][0] + ((search_space[i][1] - search_space[i][0])
   * rand())
4:   end
```

3) Evaluation generation: Fitness function evaluation is generated by translating the "evaluate("Prob_1")" DSEL specification. Translation of this DSEL construct is performed by the interpreter as illustrated in Algorithm 8.

Algorithm 8 Generation of DE evaluation in Ruby.

```
1: pop.each|c| c[:fit] = problem("Prob_1", c[:vector])
```

As can be seen from the algorithm, the fitness function is problem dependent. For instance, the implementation of the fitness function $f(\mathbf{x}) = \sum_{i=1}^{d} x_i$ referred to as "Prob_1" in DSEL specification is presented in Algorithm 9.

Algorithm 9 Implementation of DE evaluation in Ruby.

```
1: def prob_1(vector)
2:   return vector.inject(0.0) |sum, x| sum + (x ** 2.0)
3: end
```

4) Termination condition generation: The "term("gen")" DSEL specification is translated to Ruby source code by the interpreter as presented in Algorithm 10, where the first statement initializes the variable *gen*, while the second statement demands a start of the evolution cycle. However, the program needs to be finished after running 200 generations.

Algorithm 10 Generation of termination condition in Ruby.

```
1: gen = 1000
2: gen.times do |gen|
```

5) Variation operators generation for DE: Translation of the variation operators is more complex, because it consists of translating two DSEL specification: "term("Xover")" and "term("Mut")". The former specification determines the probability of the crossover operator, while the latter defines the characteristics of the mutation strategy. The corresponding generation code with appropriate parameters is presented in Algorithm 11.

Algorithm 11 Generation of operators in Ruby.

```
1: trial = oper(pop, search_space, "rand1bin", 0.5, 0.9)
```

The translated code demands a calling the mutation DE strategy 'rand/1/bin'. The implementation code of this strategy is presented in Algorithm 12.

Algorithm 12 Translation of the feature *Oper* into Ruby.

```
def de_rand_1_bin(pop, search_space, f, cr)
  trial = []
  pop.each_with_index do |p0, i|
    p1, p2, p3 = rand(pop.size), rand(pop.size), rand(pop.size)
    p1 = rand(pop.size) until p1 != i
    p2 = rand(pop.size) until p2 != i and p2 != p1
    p3 = rand(pop.size) until p3 != i and p3 != p1 and p3 != p2
    sample = :vector=>Array.new(p0[:vector].size)
    p1a = pop[p1]
    p2a = pop[p2]
    p3a = pop[p3]
    cut = rand(sample[:vector].size-1) + 1
    sample[:vector].each_index do |ii|
    sample[:vector][ii] = p0[:vector][ii]
    if (ii==cut or rand() < cr)
      v = p3a[:vector][ii] + f * (p1a[:vector][ii] - p2a[:vector][ii])
      v = search_space[ii][0] if v < search_space[ii][0]
      v = search_space[ii][1] if v > search_space[ii][1]
      sample[:vector][ii] = v
    end
    trial.push(sample)
  end
  return trial
end
```

6) Replacement generation for DE: The replacement operator is specified by the ”repl(”one-to-one”)” DSEL specification that is translated into Ruby source code as presented in Algorithm 13.

Algorithm 13 Generation of DE replacement in Ruby.

```
pop = repl(pop, trial, ”one-to-one”)
pop.sort!|x,y| x[:fit] <=> y[:fit]
best = pop.first if pop.first[:fit] < best[:fit]
puts ” > gen #gen+1, fitness=#best[:fit]”
```

The implementation of the one-to-one replacement is straightforward as can be seen in Algorithm 14.

Algorithm 14 Implementation of DE replacement in Ruby.

```
def onetoone(pop, trial)
  return Array.new(pop.size) do |i|
    (trial[i][:fit]<=pop[i][:fit]) ? trial[i] : pop[i]
  end
end
```

IV. Preliminary results

This section represents the preliminary results of generating stochastic nature-inspired population-based algorithms using the DSEL interpreter in Ruby. In line with this, two experiments were conducted as follows. In the former, the source code of the DE algorithm was simply assembled from the translated source code parts discussed in Section III, while in the latter presents the source code of the BA algorithm translated from the DSEL specifications in Algorithm 4.

The source code of DE generated from DSEL specifications in Algorithm 3 is depicted in Algorithm 15, while the DSEL specifications in Algorithm 4 produced by the source code for the BA algorithm is presented in Algorithm 16.

Algorithm 15 Main program for running simple DE

```
 1: d = 20
 2: np = 100
 3: search_space = Array.new(d) |i| [-5, +5]
 4: pop = Array.new(np) |i| :vector=>init(”Rand”, search_space)
 5: pop.each|c| c[:fit] = problem(”Prob_1”, c[:vector])
 6: gen = 1000
 7: gen.times do |gen|
 8:   trial = oper(pop, search_space, ”DE_rand1bin”, 0.5, 0.9)
 9:   trial.each|c| c[:fit] = problem(”Prob_1”, c[:vector])
10:   pop = repl(pop, trial, ”one-to-one”)
11:   pop.sort!|x,y| x[:fit] <=> y[:fit]
12:   best = pop.first if pop.first[:fit] < best[:fit]
13:   puts ” > gen #gen+1, fitness=#best[:fit]”
14: end
15: puts ”Best: #best[:fit]”
```

Algorithm 16 Main program for running simple BA

```
 1: d = 20
 2: np = 100
 3: search_space = Array.new(d) |i| [-5, +5]
 4: pop = Array.new(np) |i| :vector=>init(”Rand”, search_space)
 5: pop.each|c| c[:fit] = problem(”Prob_1”, c[:vector])
 6: gen = 1000
 7: gen.times do |gen|
 8:   trial = oper(pop, search_space, ”BA_Levy”, 0.5, 0.0, 1.0, 1.0)
 9:   trial.each|c| c[:fit] = problem(”Prob_1”, c[:vector])
10:   pop = repl(pop, trial, ”one-to-one-cond”, 0.5)
11:   pop.sort!|x,y| x[:fit] <=> y[:fit]
12:   best = pop.first if pop.first[:fit] < best[:fit]
13:   puts ” > gen #gen+1, fitness=#best[:fit]”
14: end
15: puts ”Best: #best[:fit]”
```

As can be seen in Algorithms 15-16, when comparison between each other is made both source codes are very similar. Actually, both source codes differ in calling the variation (line 8) and replacement (line 10) operators. This finding agrees with our assumption that the gap between some real-coded EAs and SI-based algorithms is not so big and that the main difference pertains mostly to the variation operator.

V. Conclusion

The main message of this preliminary study is that the internal structure of some real-coded EAs and SI-based algorithms is in fact not too different. However, more real-coded SI-based algorithms as well as evolution strategies [1] (i.e., a kind of real-coded EA) need to be analyzed in order to prove this fact in general. On the other hand, this framework also does not capture any adaptation and hybridization features.

However, at the moment, the proposed framework enables automatic generation of stochastic real-coded nature-inspired population-based algorithms and, thus, opens a new view on the development of the so-called ”new nature-inspired algorithms” that, in fact, differs between each other in the implementation of variation operators. On the other hand, this framework enables developers to mix the proposed features of the analyzed nature-inspired algorithms and thus develop new hybrid algorithms easily. The framework is also problemindependent, because solving the new problem means only an implementation of the new fitness library function.

In the future work, more stochastic nature-inspired population-based algorithms should be analyzed and incorporated into the framework. Features should also be considered like adaptation and hybridization.

References

[1] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK, 1996.

[2] Jason Brownlee. *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011.

[3] Charles Darwin. *The origin of species*. John Murray, London, UK, 1859.

[4] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: a survey of the state-of-the-art. *Evolutionary Computation, IEEE Transactions on*, 15(1):4–31, 2011.

[5] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, London, 2th edition, 2015.

[6] Iztok Fister, Iztok Fister Jr., Xin-She Yang, and Janez Brest. A comprehensive review of firefly algorithms. *Swarm and Evolutionary Computation*, 13:34–46, 2013.

[7] Iztok Fister Jr., Marjan Mernik, Janez Brest, and Iztok Fister. Design and implementation of domain-specific language easytime. *Computer Languages, Systems & Structures*, 37(4):151–167, 2011.

[8] Iztok Fister Jr, Xin-She Yang, Iztok Fister, Janez Brest, and Dušan Fister. A brief review of nature-inspired algorithms for optimization. *Elektrotehniški vestnik*, 80(3):116–122, 2013.

[9] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4), December 1996.

[10] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[11] Uros Mlakar, Iztok Fister Jr., and Iztok Fister. Hybrid self-adaptive cuckoo search for global optimization. *Swarm and Evolutionary Computation*, 2016. doi:10.1016/j.swevo.2016.03.001.

[12] Ferrante Neri and Ville Tirronen. Recent advances in differential evolution: A survey and experimental analysis. *Artif. Intell. Rev.*, 33(1-2):61–106, February 2010.

[13] Singiresu S. Rao. *Engineering Optimization: Theory and Practice*. John Willey & Sons, 4th edition, 2009.

[14] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[15] Kenneth Sörensen. Metaheuristics–the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[16] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[17] Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 10(1):1–17, 2002.

[18] Xin-She Yang. A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pages 65–74. Springer, 2010.

[19] Xin-She Yang and Suash Deb. Cuckoo search via lévy flights. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214. IEEE, 2009.