A Symbolic Evolutionary Algorithm Software Platform

Rodolfo Ayala Lopes
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais - Brazil
rodolfo.ufop@gmail.com

Thiago Macedo Gomes
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais - Brazil
thiagomacg@ufop.edu.br

Alan Robert Resende de Freitas
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais - Brazil
alandefreitas@iceb.ufop.br

ABSTRACT

Evolutionary Algorithms (EAs) have become a well-recognized population metaheuristic. The flexibility of EAs is the primary characteristic of its broad domain of practical applications. By contrast, its flexibility made it challenging to design formal languages to represent optimization models. Modeling Languages, in especial, are useful high-level languages for compact formulation and description of optimization problems. However, the lack of integration between EAs and modeling languages can delay the development of sophisticated and advanced generalized symbolic EAs, including gray box algorithms. Thus, this paper presents a Symbolic Evolutionary Algorithm Software Platform which proposes a modeling language: Procedural Modeling Language (PML). This software platform has a particular link to a symbolic compiler that allows the creation of sub-models in real-time. In the course of this paper, a study case is used to exemplify our proposed platform.

CCS CONCEPTS

• **Theory of computation** → Bio-inspired optimization;

KEYWORDS

Symbolic Evolutionary Algorithm, Software Platform, Modeling Languages

ACM Reference Format:
Rodolfo Ayala Lopes, Thiago Macedo Gomes, and Alan Robert Resende de Freitas. 2019. A Symbolic Evolutionary Algorithm Software Platform. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3319619.3326828

1 INTRODUCTION

Evolutionary Algorithm (EA) is a well known population-based metaheuristic to solving various optimization problems. Inspired on mechanisms of natural evolution, scientists proposed the first EAs in the 1960s. EAs are efficient tools for multiple fields of finance, economics, government, engineering, and science. For instance, EAs already have been applied to Brachytherapy treatment planning [19], Pharmaceutical Drug Design [17], Industrial Applications [24, 30], and Scheduling Applications [8, 26].

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326828

EAs are also useful to many categories of optimization problems, such as dynamic, robust, interactive, and multi-objective problems. This flexibility comes at the cost of having only a few formal languages to represent these models. Specific Algebraic Modeling Languages (AML), however, tend to be used in most fields of optimization as they allow for simplified mathematical notation, collaboration, and a more systematic comparison of algorithms.

Mathematical modeling of optimization problems is composed of a set of mathematical relationships, e.g., equalities, inequalities, logical conditions [13]. In order to design an abstract representation of the mathematical models, modeling languages support four basic concepts: (i) decision variables (continuous, binary, integer); (ii) constraints (equalities, inequalities); (iii) auxiliary data, also called parameters or model constants; and (iv) objective functions.

In this context, modeling languages allow the user to model the mathematical problem independently from the solver. The independent problem formulation, as well as the direct reformulation of the problem representation, are some of the main advantages of modeling languages. This independent formulation has two primary benefits: the possibility of developing more generalized solvers and that different solvers can handle the same abstract model. Furthermore, one can easily apply changes or reformulations to the abstract model, and it does not imply changes in the solver.

Modeling languages increase the optimization solvers lifetime and contribute to the reduction of the project time and maintenance [14]. Solvers widely apply this type of optimization modeling language to several types of optimization problems. The most frequently used ones are AIMMS [1], AMPL [10], CMPL [25], GAMS [2], GNU MathProg [20], MiniZinc [21], and MPL [16].

Although there are many algebraic modeling languages in the literature, there is a lack of formal languages to represent the vast optimization problem domains that EAs can solve. The difficulty of representing several practical black-box optimization problems can explain this lack of formal languages for EAs. It is essential to state that this is an obstacle even to AMLs non-free use licenses.

On the other hand, even though there have been valuable advances made on EAs over the last years, the most innovative EAs [3–6, 28] require interpretation and examination of the problem definition. However, there is still a lack of evolutionary approaches that can automatically analyze and extract information from optimization problems in a nonheuristic fashion. The absence of independent modeling languages connected with EAs can explain this lack of formal approaches. Moreover, mathematical manipulation of optimization problems is not trivial, given that the problems are usually implemented using procedural computer programming languages.

Thus, EAs integrated with modeling languages can be an interesting approach in which an automatic analysis of the problem

description would be possible. The immediate benefit of this integration is the possibility to extract an algebraic representation of the optimization problem. By facilitating the representation of the problem's algebraic properties, significant mathematical manipulations, such as simplifications, differentiation, integration, commutativity, and non-commutativity, can be easily used. This strategy can extract information about the abstract model, such as variables and constants of the model; constraints and objective functions; and, in its turn, allow inferences about decision variable interactions. Furthermore, recent EAs operators [3–6, 28] based on decomposition and variable interactions become applicable if EA-based solvers can algebraically represent and manipulate the optimization problems.

Thereby, this paper presents an Evolutionary Algorithm Software Platform that uses a modeling language that can be compiled to numeric and symbolic representations of the problem. The symbolic representation allows the formulation of domain-specific operators while the numeric representation allows fast evaluation of the objective functions. Although our approach is still embryonic, this paper presents a promissory platform toward a generalized gray box EA.

Our proposal uses a flexible modeling language, called Procedural Modeling Language (PML), to describe the model problem. The model problem is interpreted and its objective functions are compiled Just-In-Time (JIT). The JIT procedure attaches the model to a symbolic component, which is responsible for the automatically extracting an algebraic representation of the optimization problem. Thus, all information extracted is kept available for the EA during the evolutionary search.

Throughout the article, an example optimization problem is presented and is used as a study case to exemplify the different platform stages.

The remainder of this paper is organized as follows: Section 2 presents related works describing other platforms and frameworks; Section 3 presents our Procedural Modeling Language for describing optimization problem models; Section 4 describes the computational architecture of the Symbolic Evolutionary Algorithm Software Platform proposed; Section 5 describes the solver as an independent component of the architecture; and, finally, Section 6 presents a final discussion and the future works related.

2 RELATED WORKS

At the beginning of the EAs research, the development, and application of these metaheuristics have been realized ad-hoc to the treating problem. However, the broad range of application domains has induced the development of computational tools for EAs over past years. There are now platforms, frameworks and computational libraries for EAs in the literature [7, 9, 12, 15, 18, 23, 29].

However, some gaps make the development of a generalized gray box EAs and their recent EAs operators more complex. The first gap is about a common formal modeling language for describing problems. The second one is an automatic process to extract and to analyze algebraic representations of the model. Moreover, we can mention that some platforms/frameworks do not allow the inclusion of new operators.

In the context of platforms, frameworks and computational libraries for EAs, Genetic and Evolutionary Algorithm Toolbox (GEATbx) [23] is a standard tool for Matlab software. In GEATbx, the optimization problem model is declared using the Matlab language. The genome representation of the solution must be homogeneous using one of the three primitive data types (binary, integer and real). However, it does not offer immediate support for heterogeneous genotypes. By default, the GEATbx tool offers some EAs mechanisms, such as mutation, recombination, crossover, and multi-populational strategies. The main disadvantages of this approach are the non-free license and the fact that it is not extendable with new components.

The Evolving Objects library (EOlib) [15] is a C++ object-oriented framework for EAs. The EOlib supports homogenous genomic representation using primitive C++ data types as well as writing our data types. Its main advantages are: (i) the several algorithm paradigms and selection/replacement operators offered, (ii) it is open-source and free to use under GPLv2 and (iii) it is extendable to other projects.

Proposed by [7], the EAsy Specification of Evolutionary Algorithms (EASEA) is a platform dedicated to the specification of EAs. The EASEA platform also provides the exploit of parallel EAs strategies and implementations. However, this platform has its modeling language that it is not widely used and it is different from other traditional languages.

Furthermore, there are other EA libraries proposed over the past years. For instance, ECJ [18], Jenetics [29], jMetal [9], and MOEA [12] are some libraries found in the literature. In general, the main aim of the libraries mentioned is supporting the development and study of evolutionary metaheuristics.

Even though there are different platforms, frameworks, and libraries for EAs in the literature, there are gaps that prevent them from supporting gray box EAs. Moreover, they are usually not accessible to final users. The project time, evolution and maintenance of the approaches mentioned are still slow. Consequently, the lifetime of these approaches is small when comparing them with recognized solvers, such as Cplex, COIN, and Gurobi.

3 PROCEDURAL MODELING LANGUAGE

Most AMLs are not able to represent a vast number of problem categories. Moreover, when they present resources to describe several optimization problems, they do not have a free and open-source license, e.g., AIMMS and AMPL.

In this paper, we propose a modeling language called *Procedural Modeling Language* (PML). It is designed to be able to model different problems while also offering the possibility of extracting numerical and algebraic representations. The purpose of the PML is to be accessible and efficient to allow its application to a full number of optimization problems. Thus, one can readily model linear, quadratic, nonlinear, smooth nonlinear, robust, dynamic, and multi-objective optimization problems.

Following the same rationale of other modeling languages, PML suggests that the user describes a model by the following components: parameters of the model, decision variables, objective functions, and constraints. The main difference between our proposed language and the other ones is how we describe objective functions and constraints. In PML, objective functions and constraints are described using C++. Modeling functions and constraints in C++

allowed the development of a Just-In-Time (JIT) compiler based on LLVM/Clang. This JIT compiler provides an opportunity to join the user code to other C++ code or any third-party library.

In order to achieve the modeling of several categories of optimization problems, the PML supports three primary data (variables and parameters) types: integer, floating-point, and boolean. Thus, these data types supported by the PML provides resources to model binary, integer, continuous and mixed optimization problems.

Besides the mathematical description of functions, a symbolic library helps the execution of the procedural functions by extracting algebraic information from the functions. In order to exemplify the PML proposed, consider a simple optimization problem described by Equation 1, where $\mathbf{x}$ represents a solution, and d denotes the dimensionality of the problem.

$$\begin{aligned} \arg\min_{x} f(\mathbf{x}) &= \sum_{i=0}^{d-1} h(\mathbf{x}, i) \\ h(\mathbf{x}, i) &= \begin{cases} x_i x_{mod(i+1,d)}, & \text{if } mod(i,3) = 0 \\ x_i^2, & \text{otherwise} \end{cases} \\ \text{subject to } & x_i \in [-10.0; 10.0] \end{aligned} \tag{1}$$

This function can be numerically defined in C++ as shown in the Figure 1.

```
double problem(double x[], int d){
    double sum = 0.0;
    for(int i = 0; i < d; i++){
        if( i % 3 == 0 ){
            sum += x[i] * x[(i+1) % d];
        } else {
            sum += x[i] * x[i];
        }
    }
    return sum;
}
```

Figure 1: A C++ implementation of the optimization problem exemplified in Equation 1.

Given the optimization problem presented in Equation 1, Figure 2 presents the modeling of this optimization problem using the PML. Note, in this example, that a constant integer parameter d was defined as 7, representing the problem dimensionality. Next, a decimal (double) array of decision variables, called $\mathbf{x}$, was defined with size equal to 7. Moreover, the interval value of $x_i \forall i \in [0, d-1]$ is $[-10.0; 10.0]$. At the end of the model, the abstract representation of the optimization problem is modeled as a minimization objective function, using C++.

Therefore, the function in Equation 1 can also be represented by the equivalent additively separable function in Equation 2.

$$f(\mathbf{x}) = x_0 x_1 + x_1^2 + x_2^2 + x_3 x_4 + x_4^2 + x_5^2 + x_6 x_0 \tag{2}$$

For better understanding the PML, let us consider a combinatorial optimization problem, the 0-1 knapsack problem, as defined in Equation 3.

```
// defining parameters of model
int d = 7
// defining decision variables
double[-10.0, 10.0] x[7]
// defining objective functions
minimize problem = {
    double sum = 0.0;
    for(int i = 0; i < d; i++){
        if( i % 3 == 0 ){
            sum += x[i] * x[(i+1) % d];
        } else {
            sum += x[i] * x[i];
        }
    }
    return sum;
}
```

Figure 2: Modeling of the optimization problem (Equation 1) using the Procedural Modeling Language (PML).

$$\begin{aligned} \arg\max_{y} f(\mathbf{y}) &= \sum_{i=0}^{n-1} v_i * y_i \\ \text{subject to} & \\ & \sum_{i=0}^{n-1} w_i * y_i \leq max\_weight \\ & y_i \in \{0, 1\} \end{aligned} \tag{3}$$

In Equation 3, y_i represents if item i will be included or not in the knapsack, v_i represents the gain of the i-th item, and w_i represents the weight of the i-th item. Moreover, max_weight defines the maximum weight capacity of the knapsack and n defines the number of items. Thus, Figure 3 presents an example of the 0-1 knapsack model using the PML.

```
// defining parameters of model
double v = {1.2, 4.7, 2.3}
double w = {0.8, 0.5, 1.1}
double max_weight = 2.0
int n = 3
// defining decision variables
bool y[3]
// defining objective functions
maximize knapsakcProblem = {
    double sum = 0.0;
    for(int i = 0; i < n; i++){
       sum += v[i] * y[i];
    }
    return sum;
}
// defining constraint
constraint contraintWeight = {
    double current_weight = 0.0;
    for(int i = 0; i < n; i++){
       current_weight += w[i] * y[i];
    }
    return current_weight <= max_weight;
}
```

Figure 3: Modeling 0-1 knapsack problem using the Procedural Modeling Language (PML).

Concerning model changes, the PML is a very flexible modeling language. This language allows us to easily modify the problem dimensionality and variable boundaries without any complexity

increase of the model representation. This characteristic is an essential aspect of modeling languages, especially when modeling large scale optimization problems due to their vast number of decision variables and complex objective functions.

Throughout the following subsections, we present more details of the proposed modeling language, PML.

3.1 Comments

In PML, one-line comments can be preceded by `//`, `--`, and `#`. Block comments are defined with `/*` and `*/`. Figure 4 presents an example of including comments of all types.

```
// one-line comment example one
- one-line comment example two
# one-line comment example one
/* block comment example */
```

Figure 4: Example of writing one-line and block comments using Procedural Modeling Language (PML).

3.2 Parameters

Parameters represent constants or auxiliary data of the model. They are always defined by a data type, a variable name, and a constant value. In the case of arrays, a list of constant values can be defined in brackets.

Figure 5 exemplifies how to declare parameters in our proposed language. Observing this example, note that three parameters are defined, whose identifiers are *par_integer*, *par_decimal*, and *par_boolean*. In this example, the types of the parameters defined are *int*, *double*, and *bool* and their values are *7*, *5.9*, and *true*, respectively.

```
// defining parameters
int par_integer = 7
double par_decimal = 5.9
bool par_boolean = true
```

Figure 5: Example of defining parameters using Procedural Modeling Language (PML).

Similarly to declaring single parameters, the PML also allows defining arrays of parameters. The set of values must be declared between brackets and separated commas, see the example in Figure 6. Three arrays of parameters (*int*, *double*, and *bool*, respectively) are declared using the following identifiers: *array_par_ integer*, *array_par_decimal*, and *array_par_boolean*. The first array, *array_par_integer*, is composed by the set of values *10*, *20* and *30*. The values set for *array_par_decimal* are *5.893*, *7.064* and *82.09*. The *array_par_bool* is set to *true*, *false* and *true* values.

```
// defining array of parameters
int array_par_integer = {10, 20, 30}
double array_par_decimal = {5.893, 7.064, 82.09}
bool array_par_boolean = {true, false, true}
```

Figure 6: Example of defining parameters array using the Procedural Modeling Language (PML).

3.3 Decision Variables

Decision variables let us represent the solution space and their domain. Decision variables define the domain to be explored by the solver. Thus, they have constant bounds that should be respected. For instance, "`int[4,9] x`" declares a decision variable with lower and upper bounds 4 and 9. When the variable boundaries are not made explicit by the user, such as in "`int x`", the maximum limits for that data type are assumed. Moreover, boolean variables do not need bounds as they are always *false (0)* or *true (1)*.

Figure 7 shows the first example of defining decision variables using the PML. This example presents the declaration of bounded decision variables whose identifiers are *var_bounded_integer*, *var_bounded_decimal* and *var_bounded_boolean*. These decision variables are defined as the following types: *int*, *double* and *bool*. For the *var_bounded_integer* and *var_bounded_decimal* variables, their lower bounds are defined as -2 and -112.22; and their upper bounds are defined as 2 and 112.22. For *var_bounded_boolean*, the possible values are $false$ or $true$. The example also defines two decision variables, *var_unbounded_integer* and *var_unbounded_decimal*. The first one is an *int* variable and the second one is a *double* variable.

```
// defining bounded decision variables
int[-2, 2] var_bounded_integer
double[-112.22, 112.22] var_bounded_decimal
bool var_bounded_boolean
// defining unbounded decision variables
int var_unbounded_integer
double var_bounded_decimal
```

Figure 7: Example of defining decision variables using Procedural Modeling Language (PML).

Similarly to arrays of model parameters, PML allows defining arrays of decision variables. In Figure 8, three arrays are declared: (i) *var_bounded_integer*, an array of integers whose size is *4* and their values are within $[0, 10]$; (ii) *var_bounded_decimal*, an array of decimals whose size is *10* and their values are within $[0.1, 1.5]$; and, *var_bounded_boolean*, an array of bools whose size is *10,000*.

```
// defining array of bounded decision variables
int[0, 10] var_bounded_integer[4]
double[0.1, 1.5] var_bounded_decimal[10]
bool var_bounded_boolean[10000]
```

Figure 8: Example of defining array of bounded decision variables using Procedural Modeling Language (PML).

3.4 Objective Functions

Objective functions express the goal of the model. In PML, a block of C++ code or a single expression defines objective functions. Objective functions need a keyword that tells the solver whether we want to minimize or maximize that function. Extra keywords allow us to make it explicit whether a problem is robust or dynamic. Also, the model can contain any number of objective functions.

Figures 9 and 10 present two examples of objective function declaration using PML. The objective function, *minFunctionExample*, presented in the first example is defined as $\arg\min f(x, y) = 2 * cos(x) + sin(y)$. On the other hand, in the second example, *maxFunctionExample*, the equation $x_0 * x_1 - x_2 * x_0$ is declared to be maximized by the solver.

```
// defining a problem as minimize function
minimize minFunctionExample = {
    return 2 * cos(x) + sin(y);
}
```

Figure 9: Defining a minimization objective function using Procedural Modeling Language (PML).

```
// defining a problem as maximize function
maximize maxFunctionExample = {
    return x[0] * x[1] - x[2] * x[0];
}
```

Figure 10: Defining a maximization objective function using Procedural Modeling Language (PML).

PML supports modeling multi-objective problems. In this case, it is only necessary to declare all functions and whether they should be minimized or maximized.

Dynamic optimization problems can be declared in PML by adding the keyword "*dynamic*" between the *minimize*/*maximize* keyword and the function name, as the example presented in Figure 11. The dynamic keyword is used to inform the solver that the relationship between decision variables and the objective function change in time. This means the evolutionary algorithm needs to implement strategies for dynamic problems, such as reevaluating candidate solutions over time.

```
// defining dynamic problems
minimize dynamic dynamicFunctionName = {
    // c++ code
}
```

Figure 11: Defining dynamic problem using Procedural Modeling Language (PML).

Analogously, robust optimization problems are defined by the "*robust*" keyword, see Figure 12. In robust problems, we need robustness against some uncertainty in the parameters of the objective function. The solver can infer an ultimate fitness value from the average, variance, or worst case analysis of reiterated evaluations of the objective function.

```
// defining robust problems
minimize robust robustFunctionName = {
    // c++ code
}
```

Figure 12: Defining robust problem using Procedural Modeling Language (PML).

3.5 Constraints

In optimization problems with constraints, the fitness of the candidate solution is important as well as if a constraint is violated. Thus, the syntax for constraints is similar to that of objective functions. However, the keyword *constraint* is used instead of *minimize*/*maximize*, according to knapsack example presented in Figure 3.

Our platform (vide Section 4) transforms each constraint declared using PML into a single equation in which the terms from the right side of the inequality are transferred to the left side with their signs reversed. Following a similar idea, an equality constraint $h(\mathbf{x}) == 0$ can be represented as two inequality $h(\mathbf{x}) \leq 0$ and $h(\mathbf{x}) \geq 0$.

3.6 Procedures

The PML allows the declaration of procedures and functions as presented in Figure 13. Using the *procedure* keyword, declaring procedures follows the same idea of describing objective functions and constraints.

```
// defining objective functions
maximize knapsakcProblem = {
    double sum = sumProcedure(v, y, n);
    return sum;
}
// defining constraint
constraint contraintWeight = {
    double current_weight = sumProcedure(w, y, n);
    return current_weight <= max_weight;
}
// defining procedure
procedure double sumProcedure(double a[], double b[],
     int n) = {
    double sum = 0.0;
    for(int i = 0; i < n; i++){
       sum += a[i] * b[i];
    }
    return sum;
}
```

Figure 13: Procedure declaration example using the Procedural Modeling Language (PML).

4 COMPUTATIONAL ARCHITECTURE

Symbolic computation has been successfully applied to engineering and science [27] as a tool to describe and work with algebraic expressions, equations, and inequalities. It provides essential resources able to simplify, solve (symbolically), evaluate (numerically), take derivatives, and integrals of algebraic expressions.

The proposed computational architecture can be divided in four main components: *modeling problem*, *interpreting model*, *compiling/interpreting problem code* and the *solver*. Figure 14 represents these components. From modeling to solving the optimization problem, we have a special link between these four main components that allow the application of symbolic operators.

The first stage is to model the optimization problem. This modeling process is a entirely independent from the optimizer, and it can be done in any text editor. The PML presented in Section 3 completes this stage. As discussed previously, the independent modeling of the optimization problems guarantees that new problems and changes can happen without impact on the solver architecture. The parsing algorithm generates an Abstract Syntax Tree (AST) of the PML model. The parser extracts all information about parameters, decision variables, and objective functions, defined in the grammar. This information is essential for the solver to allocate the appropriate computational resources.

In its turn, the symbolic compiler uses this AST to generate function and subfunction representations in machine language that can be accessed by the solver through an interface.

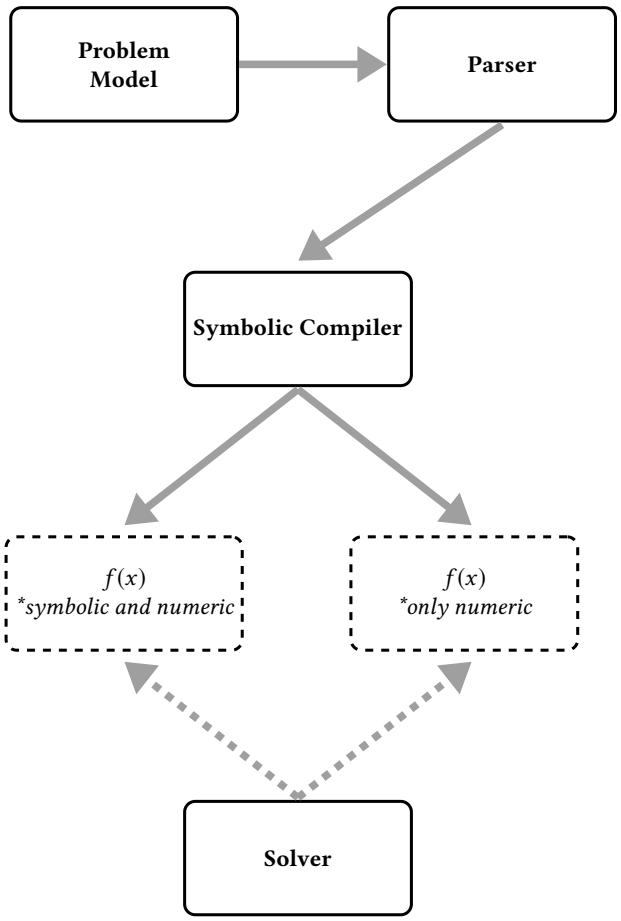

Figure 14: A brief overview of the platform architecture.

4.1 Symbolic Compiler

The primary goal of this compiler component is to provide access to efficient numeric representations of the optimization problem and its subfunctions. A symbolic library stage achieves these representations through a JIT compiler based on Clang/LLVM.

The core process of the symbolic compiler has two hierarchical procedures. At the first step, the compiler performs an automatic attempt of symbolic representation for the optimization problem. If we achieve this symbolic representation, we extract function properties and subfunctions that are useful to the solver. The compiler converts these new symbolic functions back to numerical functions that can be evaluated quickly.

For instance, the function $f(\mathbf{x})$, described in Figure 1, is converted into a symbolic representation that is simplified and then converted into an equivalent representation presented in Figure 15.

If the compiler cannot create a symbolic representation of a particular function, we directly compile it as a numerical function. Keeping the original numeric representation proposed at the modeling stage allows the solver to work on problems that cannot be easily represented algebraically, such as black-box problems, problems dependent on online data, practical dynamic and robust problems.

```
double problem(double x[]){
     return x[0]*x[1] + pow(x[1],2) + pow(x[2],2)
            + x[3]*x[4] + pow(x[4],2) + pow(x[5],2)
            + x[6]*x[0];
}
```

Figure 15: A C++ implementation of the optimization problem presented in Equation 1 when $d = 7$.

4.2 Symbolic Library

In order to represent symbolic variables, we adapted a traditional symbolic library, called *SymbolicC++*. The *SymbolicC++* library [27] uses C++ and object-oriented programming to provide an algebraic computer system which can manipulate algebraic expressions.

For this project, the *Symbolic class* from *SymbolicC++* library is its most important data type. This data type allows the algebraic representation of functions and its manipulation, such as differentiation, and integration. The *Symbolic* class represents the algebraic expressions using a tree-like structure. The original numeric representation in Figure 1 is converted into an equivalent procedural function presented in Figure 16.

```
Symbolic problem(){
    const int &d = int_parameters[0];
    Symbolic x("x", 7);
    Symbolic sum = 0.0;
    for(int i = 0; i < d; i++){
        if( i % 3 == 0){
            sum += x[i] * x[(i+1) % d];
        }
        else{
            sum += x[i] * x[i];
        }
    }
    return sum;
}
```

Figure 16: C++ code generated by the symbolic compiler component to try algebraic representation of the problem.

The result of this function is a symbolic representation of the objective function where the root node represents the identifier of the expression, the intermediate nodes represent operators and child nodes represent symbols (variables) or numeric values. Figure 17 illustrates the symbolic representation of Equation 1, where $d = 7$ and it can be represented by the equivalent as Equation 2, using the *Symbolic class*. In this example, the keywords *sum*, *prod*, and *pow* represent summation, product, and exponentiation, respectively.

Once the algebraic representation is complete, the symbolic library attempts to simplify the function and the compiler uses it to recreate the numerical function in simpler terms. This final numerical function is presented in Figure 15.

The first advantage of a symbolic approach to generate numeric functions is the cheap evaluation of objective functions achieved by not only simplifying redundant terms but also by unrolling "for" loops in the objective function by embedding values from the model parameters, such as the dimensionality of decision variables. Thus, we can translate from the general formulation in Figure 1 to an instance-specific formulation in Figure 15.

Besides generating this new and more efficient numeric representation, the symbolic compiler also generates a set of separable

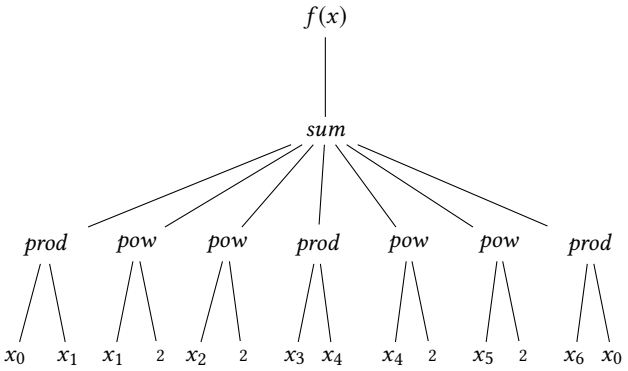

Figure 17: A tree-like structure of the function problem presented in Equation 1 when $d = 7$.

subfunctions. The ability to numerically access these subfunctions allows us to apply (i) mathematical programming methods, whenever applicable, and (ii) design EA approaches and operators based on decomposition and variable interactions.

4.3 Decomposition

Achieving the optimal decomposition approach is tightly dependent on the identification of subfunctions of the problem that can be optimized separately. Although a general procedure for identifying subfunctions is not a trivial task, we propose, in this paper, a symbolic interpretation of the problem to make it feasible. This task is possible by visiting nodes of the tree-like structure of the algebraic problem representation. This decomposition step is dependent on the concept of *separability*, as presented below.

A function $f(\mathbf{x})$, where $\mathbf{x}$ has n decision variables, is partially additively separable with m independent subcomponents if there exist a constant m such that $2 \leq m \leq n$, a set of subfunctions $f_1, ..., f_m$, and m disjoint sub-vectors of $\mathbf{x}$ denoted $s_1(\mathbf{x}), ..., s_m(\mathbf{x})$ as presented in Equation 4. Furthermore, in accordance with [22], a function is fully separable if these disjont sub-vectors are 1-dimensional, that is, $m = n$.

$$f(\mathbf{x}) = f_{s_1}(s_1(\mathbf{x})) + \ldots + f_{s_m}(s_m(\mathbf{x})) \tag{4}$$

Thus, exploring the *separability* concept, the main idea of the decomposition procedure proposed in this paper is to find additively separable subfunctions. This procedure needs to consider a symbolic interpretation of the tree-like structure of the *Symbolic class*. Then, we can visit the root node of the expression and check if its child node is a summation node. This procedure identifies that all child nodes are separable subfunctions of the problem. Otherwise, if the root node child represents any other operator, this symbol implies a non-separable subfunction.

This decomposition approach is essential to define Variable Interaction Graphs (VIG). VIGs are one of the most common approaches proposed to represent variable interactions. A VIG is an undirected graph $G = (V, E)$, where V is the set of decision variables and E represents all pairs of decision variables (x_i, x_j) that interact with each other. Each sugraph of the VIG represents a separable problem. Then, Figure 18 illustrates the VIG from the example problem in Equation 1 when $d = 7$.

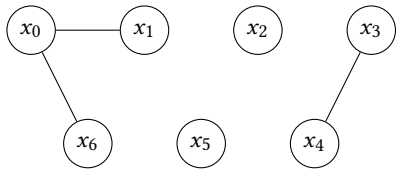

Figure 18: Exemplifying a Variable Interaction Graph (VIG) from the Equation 1 when $d = 7$.

From the VIG, the symbolic compiler component can infer two shorts subfunctions: (i) local functions and (ii) subgraph functions. Firstly, considering the problem presented in Equation 1 when $d = 7$, their local functions $f_1, f_2, \ldots, f_n$ presented in Equation 5 include only the adjacent vertices for each variable $x_0, \ldots, x_{n-1}$. Our symbolic component analyzes the code in Figure 2 and generates the equivalent C++ code accordingly as local functions described in Equation 5. The main advantage of describing local functions is that they can be evaluated at a much lower computational cost.

$$\begin{aligned} f_1(\mathbf{x}) &= x_0 * x_1 + x_6 * x_0 & f_5(\mathbf{x}) &= x_3 * x_4 + x_4^2 \\ f_2(\mathbf{x}) &= x_0 * x_1 + x_1^2 & f_6(\mathbf{x}) &= x_5^2 \\ f_3(\mathbf{x}) &= x_2^2 & f_7(\mathbf{x}) &= x_6 * x_0 \\ f_4(\mathbf{x}) &= x_3 * x_4 \end{aligned} \tag{5}$$

The subgraph functions represent separable problems from the original one. These separable problems can be defined by analyzing all vertices (variables) from a subgraph. Considering the example problem presented in Equation 1 when $d = 7$, whose VIG was described in Figure 18, their subgraph functions $(f_{s_1}, ..., f_{s_m})$ are defined in Equation 6.

$$\begin{aligned} f_{s_1}(\mathbf{x}) &= x_0 * x_1 + x_1^2 + x_6 * x_0 & f_{s_2}(\mathbf{x}) &= x_2^2 \\ f_{s_3}(\mathbf{x}) &= x_3 * x_4 + x_4^2 & f_{s_5}(\mathbf{x}) &= x_5^2 \end{aligned} \tag{6}$$

Furthermore, still observing the VIG in Figure 18, four separable subgraph functions can be identified, where x_2 and x_5 are completely independent. For instance, $f_{s_1}(\mathbf{x}) = x_0x_1 + x_1^2 + x_6x_0$ is a completely separable subfunction. This means that if $\mathbf{x}$ is a local optimum for $f(\mathbf{x})$ (and therefore, also a local optimum for $f_{s_1}(\mathbf{x})$, any solution $\mathbf{x}'$ that has the same as values for $\{x_0, x_1, x_6\}$ is also a local optima concerning $f_{s_1}(\mathbf{x})$.

5 SOLVER

According to the computational architecture presented in this paper, the solver is an independent component. This independence allows the development of different EAs without the requirement of change on the other platform components. Therefore, the solver could be composed of different EAs, e.g., Genetic Algorithm, Differential Evolution, or Genetic Programming. There is no restriction on the number of heuristics or combinations supported by the solver component. Moreover, third-party solvers could be integrated by rewriting the model in the standard modeling language of the solver.

In this platform, the solver component has access to objective functions, constraints and extras procedures through available interfaces. Furthermore, it can infer the genotype representation by analyzing the AST of the model. Thus, the information in the AST dynamically defines the representation of a solution.

Lastly, final users can use a Graphical Interface User (GUI) for modeling the optimization problem, choosing an EA, and defining parameters for the solver. Another alternative is the development of an automatic module for choosing the EA used for the problem modeled by the user.

6 CONCLUSIONS

In this paper, a Symbolic Evolutionary Algorithm Software Platform has been proposed and discussed. The computational architecture of this platform has four main components: (i) problem models; (ii) a symbolic parser; (iii) a compiler of symbolic subfunctions; and (iv) the solver. One of the objectives of this computational architecture is to provide independence between the models and the solvers. Hence, we prolong the platform life cycle.

The proposed platform has a particular link between the modeling and solver processes of the optimization problem. This particular link allows an automatic attempt of symbolic compiling which provides an algebraic representation of the problem. Although the algebraic representation of the model is conceivable for many optimization problems (e.g., linear, non-linear, combinatorial, multi-objective), the algebraic representation can be challenging or impracticable for some problems, for instance, black-box problems which are dependent on external data. Thus, for these particular cases in which one cannot algebraically represent the objective functions or constraints, only the original representation proposed at the modeling stage will be provided to the solver.

In order to try to do a symbolic compilation, we have also proposed a modeling language, called Procedural Modeling Language (PML). In PML, the objective functions and constraints are declared using C++, which allows the integration with third-party libraries. The symbolic representation provided through integration between the C++ code and the third-party symbolic library supports the algebraic representation and manipulation of the model. This algebraic representation and manipulation make it possible to advance toward a gray box optimizer.

Therefore, this software platform still needs support from a solver that takes advantage of all these capabilities. Ideally, the archetypical solver would be able to use the subfunctions identified in the model to enhance the optimization process. Recently, Gomes *et al.* [11] have successfully used this platform for the development of a multi-heap constraint handling evolutionary algorithm.

ACKNOWLEDGMENTS

We would like to thank the agencies CAPES, CNPq (402956/2016-8), FAPEMIG (00040-14), and UFOP for financially supporting the development of this research.

REFERENCES

[1] J. Bisschop. 2006. *Aimms Optimization Modeling.* Paragon Decision Technology.
[2] M. R. Bussieck and A. Meeraus. 2004. *General Algebraic Modeling System (GAMS).* Springer, Boston, MA, 137–157.
[3] F. Chicano, G. Ochoa, D. Whitley, and R. Tinós. 2018. Enhancing Partition Crossover with Articulation Points Analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM, New York, NY, USA, 269–276.
[4] F. Chicano, D. Whitley, G. Ochoa, and R. Tinós. 2017. Optimizing One Million Variable NK Landscapes by Hybridizing Deterministic Recombination and Local Search. In *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM, New York, NY, USA, 753–760.
[5] F. Chicano, D. Whitley, and A. M. Sutton. 2014. Efficient Identification of Improving Moves in a Ball for Pseudo-boolean Problems. In *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM, New York, NY, USA, 437–444.
[6] F. Chicano, D. Whitley, and R. Tinós. 2016. Efficient Hill Climber for Constrained Pseudo-Boolean Optimization Problems. In *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM, New York, NY, USA, 309–316.
[7] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. 2000. Take It EASEA. In *Parallel Problem Solving from Nature.* Springer, Berlin, Heidelberg, 891–901.
[8] M. A. Dulebenets. 2018. Application of Evolutionary Computation for Berth Scheduling at Marine Container Terminals: Parameter Tuning Versus Parameter Control. *IEEE Transactions on Intelligent Transportation Systems* 19 (2018), 25–37.
[9] J. J. Durillo and A. J. Nebro. 2011. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software* 42 (2011), 760–771.
[10] R. Fourer, D.M. Gay, and B.W. Kernighan. 2003. *AMPL: A Modeling Language for Mathematical Programming.* Thomson/Brooks/Cole.
[11] T. M. Gomes, A. R. R. Freitas, and R. A. Lopes. 2019. Multi-heap Constraint Handling in Gray Box Evolutionary Algorithms. In *In Proceedings of the Genetic and Evolutionary Computation Conference.* ACM.
[12] D. Hadka. 2016. *Beginner's Guide to the Moea Framework.* CreateSpace Independent Publishing Platform.
[13] J. Kallrath. 2004. *Mathematical Optimization and the Role of Modeling Languages.* Springer, Boston, MA, 3–24.
[14] J. Kallrath. 2012. *Algebraic Modeling Languages: Introduction and Overview.* Springer, Berlin, Heidelberg, 3–10.
[15] M. Keijzer, J. J. Merelo, G. Romero, and Marc Schoenauer. 2002. Evolving Objects: A General Purpose Evolutionary Computation Library. In *Artificial Evolution.* Springer, Berlin, Heidelberg, 231–242.
[16] B. Kristjansson and D. Lee. 2004. *The MPL Modeling System.* Springer, Boston, MA, 239–266.
[17] E. Lameijer, T. Bäck, J. N. Kok, and AD P. Ijzerman. 2005. Evolutionary Algorithms in Drug Design. *Natural Computing* 4 (2005), 177–243.
[18] S. Luke. 1998. ECJ Evolutionary Computation Library. (1998). Available for free at http://cs.gmu.edu/~eclab/projects/ecj/.
[19] N. H. Luong, T. Alderliesten, A. Bel, Y. Niatsetski, and P. A. N. Bosman. 2018. Application and benchmarking of multi-objective evolutionary algorithms on high-dose-rate brachytherapy planning for prostate cancer treatment. *Swarm and Evolutionary Computation* 40 (2018), 37–52.
[20] A. Makhorin. 2010. *Modeling Language GNU MathProg - Language Reference.* Free Software Foundation, Inc.
[21] K. Marriott, P. Stuckey, L. De Koninck, and H. Samulowitz. 2019. An Introduction to MiniZinc. (04 2019).
[22] M. N. Omidvar and X. Li. 2017. Evolutionary large-scale global optimization: an introduction.. In *In Proceedings of the Genetic and Evolutionary Computation Conference Companion.* ACM, 807–827.
[23] H. Pohlheim. 2006. GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with Matlab. (2006).
[24] E. Sanchez, G. Squillero, and A. Tonda. 2012. *Post-silicon Speed-Path Analysis in Modern Microprocessors through Genetic Programming.* Springer Berlin, 31–44.
[25] M. Steglich and T.F Schleiff. 2010. *CMPL: Coliop Mathematical Programming Language.*
[26] C. J. Tan, S. C. Neoh, C. P. Lim, S. Hanoun, W. P. Wong, C. K. Loo, L. Zhang, and S. Nahavandi. 2019. Application of an evolutionary algorithm-based ensemble model to job-shop scheduling. *Journal of Intelligent Manufacturing* 30 (2019), 879–890.
[27] K.S. Tan, W.H. Steeb, and Y. Hardy. 2000. *SymbolicC++:An Introduction to Computer Algebra using Object-Oriented Programming: An Introduction to Computer Algebra Using Object-Oriented Programming.* Springer London.
[28] R. Tintos, D. Whitley, and F. Chicano. 2015. Partition Crossover for Pseudo-Boolean Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM, New York, NY, USA, 137–149.
[29] F. Wilhelmstötter. 2019. Jenetics - Library User's Manual 4.4. (2019).
[30] J. Wodecki, A. Michalak, and R. Zimroz. 2018. Optimal filter design with progressive genetic algorithm for local damage detection in rolling bearings. *Mechanical Systems and Signal Processing* 102 (2018), 102–116.