A Semantic Model for Evolutionary Computation

Christian Veenhuis[1], Katrin Franke[2] and Mario Köppen[2]
[1]Technische Fachhochschule Berlin, University of Applied Sciences
Luxemburger Str. 10, 13353 Berlin, Germany
Email: veenhuis@fth-berlin.de
[2]Fraunhofer Institute for Production Systems and Design Technology
Department Pattern Recognition
Pascalstr. 8-9, 10587 Berlin, Germany
Fon: +49 30 39006302, Fax: +49 30 3917517, E-mail: {katrin.franke | mario.koeppen}@ipk.fhg.de

Key Words: evolutionary computation, semantic model, markup language, algorithm design

Abstract—
We propose a semantic model for evolutionary computation, which supports the rapid design of a huge variety of evolutionary-algorithm-structures and can be represented in XML. Moreover, a designed evolutionary-computation-model can also be used as compiler input to generate ready to use object-oriented code covering the adequate evolutionary-computation-functionality as well as for documentation and exchanging the realised algorithm. The basic concept of the model is founded in a such called component hierarchy, where operators and basic algorithms are handled as programmable nodes of a structural tree and where the structural tree describes the computation flow. Within this paper the description of the semantic model is presented followed by a detailed example. It will be shown how the proposed approach enables a more abstract handling of the evolutionary algorithms, and how it speeds up the algorithm design. Also, it will be presented how rapid structural changes of the evolutionary algorithm might be performed by simple changing the operator hierarchy or by the replacement of the programmable nodes.

1 Introduction

In the development as well as in the application of evolutionary algorithms (EA) it is useful to handle a specific algorithm on a more abstract level than the level of C/C++ function-calls. However, many activities are directed at the implementation of C/C++ software libraries covering EA functionality [4] [5] [6] [7] [8]. Moreover, a documentation that has to be served is usually created later on. These separated working lines are inconvenient, cause a lot of mistakes and prohibits standardization. To overcome the mentioned problems and to revival efforts in exchangement and standardization, we propose a semantic model, which supports the modeling, description and documentation of EA at once. The semantic model should be system-independent and separated from an evolved solution and its data. Also, it should be easy to understand and to learn.

Looking for a suitable concept, different approaches were studied. Among them are, **H**yper**t**ext **M**arkup **L**anguage (HTML) [1], **V**irtual **R**eality **M**odeling **L**anguage (VRML) [3] and E**x**tensible **M**arkup **L**anguage (XML) [2]. The named languages are not traditional programmer-languages that specify a computation flow. They are developed to specify and to characterize elements of an environment. Derived form the **S**tandard **G**eneralized **M**arkup **L**anguage (SGML) [2], a standard (ISO 8879) from 1986, these languages were developed to describe a wide variety of structural discriminate environments. The languages abstract several elements by adapted constructs and supports their hierachical ordering. As an example VRML is used to model geometric objects in 3D-worlds where objects like polygons are determined by such called nodes. Each node covers further parameters like coordinates and edges, which are necessary to represent a polygon. Finally, a such called node-hierarchy describes the whole 3D-world. HTML and XML, being more popular, follow a comparable concept of element-hierarchy.

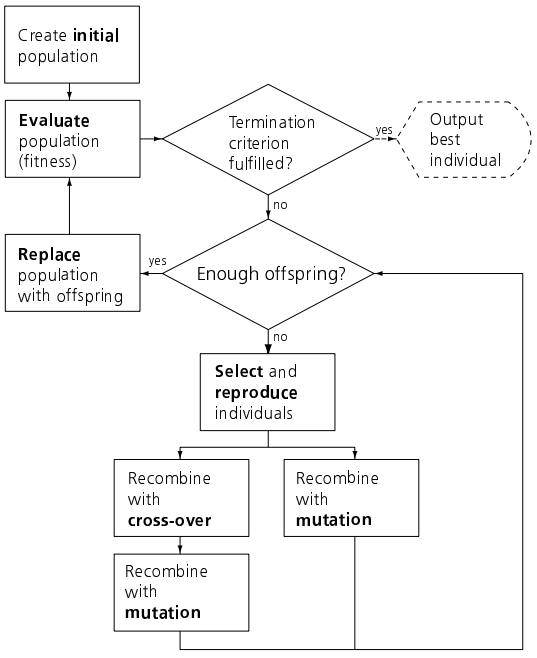

Figure 1: Flow diagram of a possible GA as an example

Currently, there are a lot of activities to set up a world wide standard for multimedia documents. XML promises to fulfill upcoming demands and researchers from different fields, esp. mathematicians, chemists, and musicians, work on the extension of the primary XML. From our point of view the EC-community should jump on the bandwagon and should use chances like this for its own evolution.

The authors propose a concept for an **E**volutionary **A**lgorithm **M**odeling **L**anguage (EAML). Several elements should be provided to describe methods and operators of EA's on an abstract level. Programmable code-elements should support the extention of the primary EAML. Finally, the proposed element-hierarchy should realize a complete EA.

By using XML-notation documentation and exchangement of an EA could be quite nice. However, there is the fundamental demand to process data with a designed EA. So, low-level function calls like C/C++ are still required. Naturally, this should be more or less invisible for a user. In VRML a scene-description is interpreted to play fantastic virtual realities online. For EA-applications a compiler-approach translating EAML-descriptions to ready to use C/C++ code seems to be more suitable.

In the section 2, we will present the actual state of EAML, the semantic model for EA's. To find out whether the created EAML is suitable for practical applications or not, an EAML-compiler, generating C++ code, was implemented in parallel and will be introduced briefly in section 3. Also, some EA-standard-problems were tested and the results will be given in section 3. Finally, section 4 gives an outlook to further developments. The authors would like to motivate other contributions for the extension of EAML as well as for standardization and exchangement within the EC-community.

2 Derivation of the semantic model

2.1 Components of an evolutionary algorithm

To model an EA its specific components will be recalled. Out of the family of EA's we choose a simple genetic algorithm (GA) as an example. The computational flow-chart of a possible GA can be seen in figure 1.

In general, a GA comprises mainly the following components:

- An *initialization rule* to generate the starting population.
- An *evaluation and fitness function* to determine the quality.
- A *stopping criterion*, which has to be fulfilled.
- A *marriage rule* to select parents for reproduction.
- *Genetic operators* (e.g. crossover, mutation) to recombine genes for the next generation.
- *A replacement scheme* to specify how the next generation is produced from offsprings.
- A *data structure* representing the encoding of an evolved solution.

The GA presented in figure 1 comprises two parallel genetic operators, whereby one line consists of a crossover operator followed by a mutation and whereby the other line is represented only by a mutation operator. In this special case, the next generation will be produced alternatively by one of both lines. For GA's most of the adaptations occur with the genetic operators whereby their kinds and their structural relations may vary.

Although, upcoming components might not be predicted, hence, the semantic model that describes EA's has to consider the huge variability of structural design and the great number of different methods and operators.

2.2 Requirements

From the presented sample (figure 1), further requirements for a semantic model can be deduced. The several components of an EA, methods as well as operators, could be handled as building blocks. So, it will be possible to tickle around with that building blocks, being simply called *elements*. Finally, a complete EA could be realized by a nested arrangement of elements. For users convenience the most common methods and operators should be predefined. Specialized methods could be integrated as additional modules. Procedures for the structural description of an EA and for the initialization should be provided. For the description of a huge variety of EA-structures, there has to be an opportunity to extent the semantical model by user-defined elements. For difficult operators or methods, code, implemented in a higher object-oriented programming language, should be inserted.

One of the most important requirements is that an EA-model has to be designed for a problem-class and not for a specific instance of a problem. So, instance-specific parameters should be adjustable later on.

Finally, a such modeled EA can be studied, exchanged and documented very easily. In the following section, we will have a look on a suitable notation fulfilling the stated requirements.

2.3 Notation and basic concepts of EAML

The notation and the basic concept of EAML follow the XML-standard [2]. In general, the structure of EAML consists of several elements and an ordered collection of such elements can be named an EA-description. Such an EA-description is stored in an ASCII-file (figure 4) and can be created with a simple text-editor. The several elements were applied in the following general notation:

```
<Elementename
  Attribute[i]name="Attribute[i]value"
> content <\Elementname>
```

Each element is introduced by a such called $<StartTag>$ and finished with an $<\backslash EndTag>$. Enclosed of this both tags the element-content is written. Depending of the element-type an attribute-list containing further parameters can be set

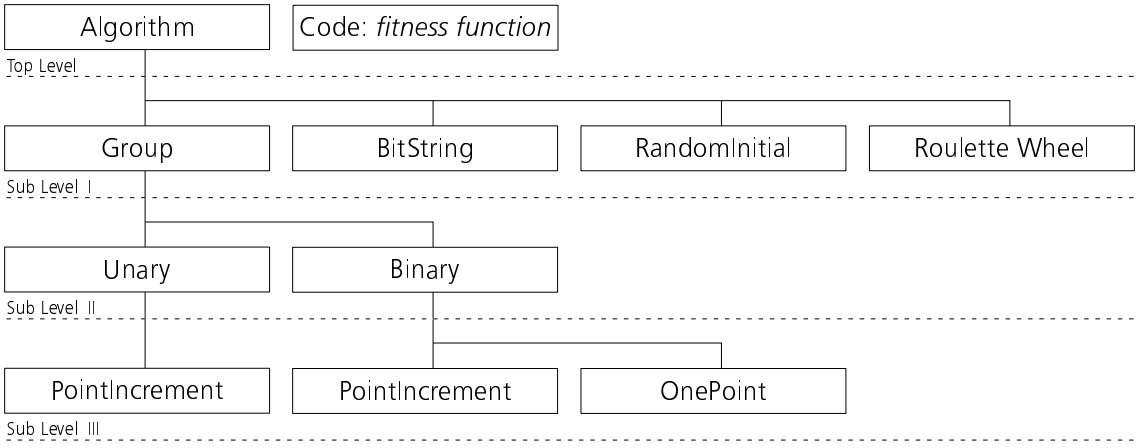

Figure 2: Component hierarchy of the sample GA

up. It has to be considered that attributes are able to carry default-values, therefore they haven't to be specified explicitly. Also, each element might contain further elements, therefore it is spoken of nested elements or an element-hierarchy.

The definition-notation for several element-types and their attributes was taken over from XML [2] and should not explained further on at this point.

Each method, like selection or fitness-evaluation, coding, the genetic operators and all other components, required for the description of an EA, has to be abstracted by the mentioned elements. So, up to now, every standard-operator and every standard-method of an EA might be handled as complete programmable building block. To give an example, a bit-string can be described by the following notation:

```
<BitString size="0" group="1"><\BitString>
```

and shortened according to the XLM-standard:

```
<BitString size="0" group="1"\>
```

The element is named by its meaning and comprises two attributes. By the *size*-attribute the length of the bit-string can be determined, by the *group*-attribute the number of bits, which are used in a unit (gene), is stated. As mentioned before, the attribute-values might be changed in a specific EA-description. Another example that should be provided is a *selection*-element that contains another element determining the selection-method:

```
<selection><RouletteWheel/></selection>.
```

By using the presented notation all components of an EA can be modeled and it is equal whether they are selection-methods, fitness-functions, genome coding or genetic operators.

2.4 The element hierarchy

A complete EA-description consists of diverse elements arranged in a nested order. So, the elements form a hierarchy, which is mentioned as element-hierarchy. This element-hierarchy also represents the affiliation of an element whereby each element of a lower level is contained by an element of the higher level. Currently, on the top-level there are following element-types permitted:

Algorithm	Covers the complete description of the EA
Random	Defines users random generator globally
Code	Contains users code, in C/C++, e.g. to provide an adapted fitness function

As suborder-elements of the *algorithm*-element, up to now, the following elements were provided. Moreover, each of these elements covers further elements by its own.

Selection method	Uniform, RouletteWheel, Greedy-Over, Tournament
Fitness function	Rawfitness, ReversalFitness, RelativeFitness, NormalizedFitness
OperatorElement	Unary, Binary, Group, Switch
BinaryOperator	OnePoint, Uniform, Cycle
UnaryOperator	PointRandom, PointIncrement, Swap, SwapNeighbor, Inversion
Initialize	RandomInitial, ConstantInitzial, IndexInizial
ReplacementScheme	Generational, DeleteNLast
Coding(Genome)	Gene, BitString, {Int,Long,Float,Char}Vector

The number of *algorithm*- and *code*-elements at the top-level is not restricted. From an *algorithm*-element hierarchically downwards all components of a particular EA are specified. The description of an EA-structure is realized by a sub-tree with the root being an *operator*-element. The sub-tree might contain any nested scopes. So, it will be possible to design a wide variety of different EA-structures. Also, on the top-level a *code*-element is permitted. The C/C++-Code implemented in that element-content has to be pluged-in by an *use*-element. From that it should be evident that an EA-

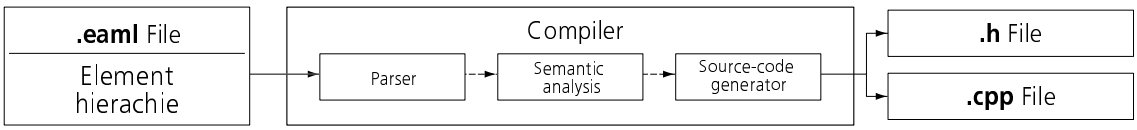

Figure 3: Compiler for the .ecml File

description can be extended and that a huge amount of EA can be modeled. However, there is still some work. So, the authors are still discussing an event/message-concept, which would support the interaction of several elements like *algorithm*-elements running parallel.

For a better understanding we will design a simple problem, for which an EA by using EAML notation will be modeled. For each problem that can be coded by a BitString an evolved solution may be found that is also represented by a BitString. To make it very clear, we assume that our problem can be solved by a BitString with $size = 30$. The value of each vector-element is assumed as $v = 1$. Also, we want to use the GA whose flow-chart is given in figure 1. By considering all components of this specific GA we can represent them by the more abstract, graphical element-hierarchy (see figure 2.) Putting all together and converting the visualized element-hierarchy into EAML we will get a description for the algorithm as it can be seen in figure 4. At the top-level we see the *algorithm*-element as well as a *code*-element that determines the fitness function.

The EAML-description in figure 4 handels the chosen sample GA at an abstraction-level that compares with the flow-chart in figure 1 or with the graphical element-hierarchy in figure 2. After "clear thinking" and finishing the modeling, the EA- description using EAML can be supplied for EA-code-generation.

3 Realisation and Results

3.1 Implementation of an EAML-compiler

Concomitant with the EAML-concept, a compiler for EAML was designed and implemented. This was necessary to prove whether the proposed semantic model is meeting practical expenses. As an outcome of the compilation, ready to use code in a higher object-oriented programmer language should be provided, which might be included in an extensive software project, or which is used to realize a standalone application covering EA functionality. For the final realization of the EA we used C++ as object-oriented programming language. The generated code matches the ANSI C++ specification. Optionally, a standalone console-program or a module for further integration can be generated.

A close view on the compiler architecture can be seen in figure 3. The element-hierarchy describing the EA is stored in the .eaml-file. This file as well as possible includes of the .eaml-file will be supplied to the compiler. The compiler processes the provided files into three stages. By parsing the element-hierarchy a syntax-tree is generated at first, followed by a semantic analysis. This analysis is necessary for consistency check. For example it has to be ensured that each .eaml-file contains at least one *Algorithm*-element. If everything is fine, the syntax-tree is forwarded to the source-code-generator. At this point the algorithm-routines, referenced by the .eaml-file and needed to realize the EA-functionality, are combined and adapted by the given parameters. Finally, the source-code-generator produces the C++ Code and provides a .h- and a .cpp-file, which might be compiled with a main-program or which might be integrated into another software project.

3.2 Application of EAML

To produce an applicable EA modeled by EAML, the required components of the EA (compare figure 1) has to be determined at first. Particularly, the genome coding has to be ventilated as well as the operators have to be selected. From this chosen elements an element-hierarchy as you can see in figure 2 has to be set up. In the following, the element-hierarchy, which also describes the interconnection of the elements, might be coded in EAML. The EAML-code-structure of our sample is presented in figure 4. Finally, the coded element-hierarchy is stored in an .eaml-file, which is then forwarded to the EAML-compiler. In case there are no mistakes, the compiler will generate the .h- and the .cpp-file. Lets assume that after testing of our produced EA we would like to adjust some parameters or we would like to exchange a selection rule or a genetic operator. For doing so, we only have to edit the element-hierarchy. Even if we have an idea for a new genetic-binary-operator, we can extend our element-hierarchy. By applying the *code*-element we implement what we need in a higher object-oriented programming language, mainly C/C++, and use it later on. In this way it will be also possible to extend the primary EAML and to adapt it to user efforts.

3.3 Sample EA's modeled with EAML

Various EA's were modeled with EAML and the EA-functionality's were provided by using the implemented compiler. Our aim was to model a well-know problem with EAML and to check whether the generated EA-functionality is able to find an evolved solution that we expected. In table 1 three sample applications are presented briefly. Of course, we also tested the traveling sales man problem (TSP). The TSP was configured with 30 "cities" and 200 individuals. For encoding, an integer vector was used holding the city-indexes

EA type	Evolutionary strategy	GA	Genetic programming
Sample	XOR problem	knapsack	symbolic regression
Description	The weights of a Perceptron with one hidden layer were adapted to solve the XOR-problem	A knapsack has to be packt with a selection of things determined by weight and value. The carry-weight of the knapsack is limited. However, the goal is to put as much things as possible in the knapsack to get a high value.	Given is a number of points P with $P(x, f(x))$. An envolved solution to approximate $f(x)$ has to be provied by the EA. We used $f(x) = \sin x^2 \cos x$ and got the evolved solution with an Error $E = 1.4409^{-10}$
Encoding	FloatVector	BitString	Tree with five levels
Population	100	100	500
Generation	30	8	35

Table 1: Samples of problems realized with EAML

and the travel-route were determined by the order of city-indexes within the vector. An evolved solution was found after 140 generations in average. As it can be seen, the proposed semantic model and its reference implementation raise EA-solutions as they were expected. Being realistically, there is a huge amount of further testing required. The authors would like to motivate other research groups to work with EAML to determine possible restrictions and to come over them by improving the semantic model.

4 Conclusion and future work

Whitin this paper a semantic model following the XML-notation (E**x**changeable **M**arkup **L**anguage) was presented that enables the modeling of evolutionary algorithms (EA). In this way it will be possible to handle EA's with their components at the abstraction-level of building blocks. The semantic model applicable by EAML (**E**volutionary **A**lgorithm **M**odeling **L**anguage) enables the modeling as well as the documentation of an EA simultanously and ensures the interchange of EA modeled once.

The notation of EAML supports the definition and integration of user-defined methods and operators, what seems to be useful for the further development and extension of EAML. Up to now, well-known EA-problems were modeled with EAML. To validate the designed EA-models an EAML-compiler was also implemented and presented within this paper. As it was shown, realized EA-functionalities have fulfilled the expectations. However, there is still some work. For example it seems to be useful to provide an *operator*-element, which is only activated on demand, like by matching a specific criterion during the iterations. Also, we are looking for mechanisms to handle sub-samples of individuals during the iterations. And, it would be quite nice to provide mechanisms for parallelization and intercommunication of several *algorithm*-elements.

Currently, an EA-description has to be typed with a simple text-editor. The authors and maybe also upcoming users of EAML would like to use an enhanced EAML-editor, or even a graphical EAML-builder, which supports the design of an graphical element-hierarchy like in figure 2. Inversely, a graphical element-hierarchy should be generated by a supplied EA-description using EAML, what would be very useful for the documentation and visualization of a modeled EA. The proposed EAML-concept, that follows the XML-standard, also supports the integration of further elements, like plain text, images or even (JAVA)-scripts.

Last but not least, the authors would like to motivate discussions and further contributions for the extension of EAML as well as for standardization and exchangement within the EC-community.

References

[1] Laura Lemay, "Teach Yourself Web Publishing with HTML in a week", SAMS Publishing, Indianapolis, Indiana, 1995

[2] Henning Behme and Stefan Mintert, "XML in der Praxis: Professionelles Web-Publishing mit der Extensible Markup Language", Addison-Wesley, Bonn, Deutschland, 1998

[3] Hans-Lothar Hase, "Dynamische virtuelle Welten", dpunkt, Verlag, heidelberg, Deutschland, 1997

[4] The GALib library <http://lancet.mit.edu/ga/>, 2000

[5] The TOLKIEN library <http://alife.ccp14.ac.uk/zooland/encore/www/ ... ...Q20_tolkien.htm>, 2000

[6] The EO library <http://geneura.ugr.es/ jmerelo/EO.html>, 2000

[7] The GAGS library <http://kal-el.ugr.es/GAGS/>, 2000

[8] The PeGAsuS project <http://set.gmd.de/AS/pega/>, 2000

```
<?xml version="1.0"?>

<EAML  standalone="true"  project="bits">

<!-------------------------------------------------------------------------------------------------------->
<Code name = "fitfunc">
                                        // this is the objective-function
  double fit = ECPARAM(CBits,gene,size);  // the worst fitness is the number of bits

  // Compare whole genome bitwise with desired target.
  // Every equal bit-pair causes a decrementation.
  // Thus the best fitness is 0 and the worst is the number of bits within genome.

  for (int i = 0 ; i < ECPARAM(CBits,gene,size) ; i++ )
      if ( GETDATA( i ) == 1 )  fit -= 1;
  return fit;

</Code>

<!-------------------------------------------------------------------------------------------------------->
<Algorithm  name = "CBits"              <!-- the main-algo (we use only one) -->

  size = "20"                           <!-- size of population = number of individuals -->
  direction = "minimize"                <!-- we want to minimize the objective-function -->
  generations = "100"                   <!-- run max. 100 generations -->
  optimum = "0.0"                       <!-- this is our target what we want to reach -->

  elitistRate  = "10%"                  <!-- we use 2 elitists (20 * 0.1 = 2) -->
  operatorRate = "90%"                  <!-- create rest of new population with operators -->
>
  <objective><Use ref="fitfunc"/></objective>          <!-- reference to objective-function -->

  <genome><BitString size="30" group="1"/></genome>    <!-- use every bit alone -->

  <selection><RouletteWheel/></selection>              <!-- fitness-proportional selection -->


  <operator><Group><operator>                          <!-- This is the operator-structur. -->
                                                       <!-- We create two parallel operators: -->
      <Binary rate = "90%" succRate = "10%" >          <!-- a binary (cross-over p=0.9) and -->
                                                       <!-- an unary (mutation p=0.1). -->
        <method><OnePoint/></method>                   <!-- After creating offspring with the binary -->
                                                       <!-- operator, we additionally mutate the -->
                <succ type = "unary">
                            <PointIncrement            <!-- offspring with p = 0.1. -->
                               min="0"
                               max="1"
                               step="1"
                            />
                </succ>
      </Binary>

      <Unary rate="10%"><method><PointIncrement min="0" max="1" step="1"/></method></Unary>

  </operator></Group></operator>                       <!-- end of operator-structur -->
  <initial><RandomInitial min="0" max="1"/></initial>  <!-- initialize the bitstrings randomly -->

</Algorithm>
</EAML>
```

Figure 4: A sample element hierachy written in EAML