A Genetic Programming Problem Definition Language Code Generator for the EpochX Framework

Claris Leroux
INSA Rouen
Haute-Normandie
France
claris.leroux@insa-rouen.fr

Fernando E. B. Otero
School of Computing
University of Kent
Chatham Maritime, UK
F.E.B.Otero@kent.ac.uk

Colin G. Johnson
School of Computing
University of Kent
Canterbury, UK
C.G.Johnson@kent.ac.uk

ABSTRACT

There are many different genetic programming (GP) frameworks that can be used to implement algorithms to solve a particular optimization problem. In order to use a framework, users need to become familiar with a large numbers of source code before actually implementing the algorithm, adding a learning overhead. In some cases, this can prevent users from trying out different frameworks. This paper discusses the implementation of a code generator in the EpochX framework to facilitate the implementation of GP algorithms. The code generator is based on the GP definition language (GPDL), which is a framework-independent language that can be used to specify GP problems.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

EpochX, genetic programming, GPDL, code generator

1. INTRODUCTION

EpochX [6] is an open source genetic programming (GP) framework, which provides three flavours of GP representations: strongly-typed tree GP (STGP) [1, 3], context-free grammar GP (CFG-GP) [9], and grammatical evolution (GE) [5]. The framework has a modular and flexible architecture that allows the customization of almost every aspect of a GP algorithm and, at the same time, it provides default implementations of popular procedures (e.g., roulette/tournament selection, grow/full/ramped half-and-half tree initialisation). It also provides an event-based framework to track the progress of the algorithm, allowing users to register listeners that receive notifications about its lifecycle (e.g., at the start/end of a generation, when a particular genetic operator is used). The information provided by the events can also be used to generate elaborated output, such as GUI interfaces [7], without requiring any extra implementation in the GP algorithm.

While EpochX structure is clear and the framework provide base classes to help the definition/implementation of new algorithms, a user has to become familiar with their purposes and uses. For new users, coming from a different framework, there might not be much similarities that can reduce the effort of implementing an algorithm in EpochX. Kronberger et al. [2] identified that implementing GP algorithms using specific frameworks involves writing a lot of code that is not directly related to the problem specification, but required to fit the problem implementation into the framework. The Genetic Programming Problem Definition Language (GPDL), proposed in [2], is designed as alternative to specify GP problems with the aim of reducing the effort of implementing GP algorithms. This paper presents a code generator for the EpochX framework that takes a GPDL problem definition to generate the required source code to run the GP problem in the EpochX framework. The motivation is to reduce the effort of writing GP algorithms using EpochX, since most of the framework-specific details are automatically generated by the code generator and the user only needs to focus on the GPDL problem description.

The remainder of the paper is structured as follows. Section 2 presents an overview of the genetic programming problem definition language, which is the language used to specify the GP problem to the code generator. Section 3 presents the details of the code generator. Section 4 presents a complete example of a GP problem definition written in GPDL for the EpochX framework. Finally, Section 5 concludes the paper and presents future research directions.

2. GPDL

The Genetic Programming Problem Definition Language (GPDL) was proposed by Kronberger et al. [2] as a framework-independent language to specify GP problems. One of the main motivations of the GPDL proposal was to reduce the effort of using a GP framework to implement new GP algorithms. As discussed by Kronberger et al., the implementation of a GP algorithm requires a lot of boilerplate code, which most of the time is very repetitive—e.g., in object-oriented GP frameworks, the definition of a new non-terminal (function) involves the sub-classing of a base non-terminal class, in combination to adding package imports and other specific language requirements. On top of this, the user of the GP framework has to become familiar with a large amount of framework-specific source code—since frameworks generally have a very different API and can be implemented in different programming languages—instead of focusing only on the problem definition.

A problem definition written in GPDL allows the user to specify all details of the GP problem in a single file, including the definition of the syntax (e.g., terminal and non-terminal symbols, grammar rules) and the semantics (e.g., non-terminal behaviour, fitness function). The main sections in a GPDL problem definitions are:

- `PROBLEM <name>:` Begins the problem definition and it is followed by the name of the problem;
- `CODE:` Allows the inclusion of additional source code—e.g., variable declarations;
- `INIT:` Allows the inclusion of additional source code that is run only once at the beginning of the execution;
- `NONTERMINALS:` Contains the definition of non-terminal symbols, including its name, parameters and semantic action (behaviour);
- `TERMINALS:` Contains the definition of terminal symbols, including its name, semantic action and constraints (e.g., values range);
- `RULES:` Grammar rules of the problem, using the terminals and non-terminals defined previously;
- `MAXIMIZE` or `MINIMIZE:` The objective (fitness) function to be either maximized or minimized.

While the GPDL is independent of the framework and programming language, a problem definition specified using the GPDL will eventually be dependent on the specific framework and programming language, as can be seen in Section 4. This is due to the fact that the GPDL allows and requires the specification of the semantics (behaviour) and other source code fragments in the target programming language, and therefore, the problem definition will have non-standard code. This arguably can be seen as a weakness of the GPDL, but it does not take the usefulness of having problem definitions in GPDL. It is much easier and quicker to convert the framework-specific problem definition to be used in a different framework than to re-write the whole source code.

3. IMPLEMENTING GPDL SUPPORT IN EPOCHX

The first requirement in order to support GPDL in a GP framework is to create a GPDL compiler. The compiler is created using the Coco/R compiler generator [4]. Coco/R takes the GPDL syntax definition[1] as an input, which is an attribute grammar (ATG) file describing the lexical and syntactical structure of the language as well as its semantic actions for the target framework, and produces a scanner and parser for the GPDL language. The scanner is responsible for reading tokens from the input buffer. The parser is responsible for processing the tokens and generating the source code (implementation) of the problem definition, according to the grammar definition and semantic actions. This process is illustrated in Figure 1. It should be noted that the scanner and parser generation only need to be performed once and it should be repeated in case of a new GPDL specification.

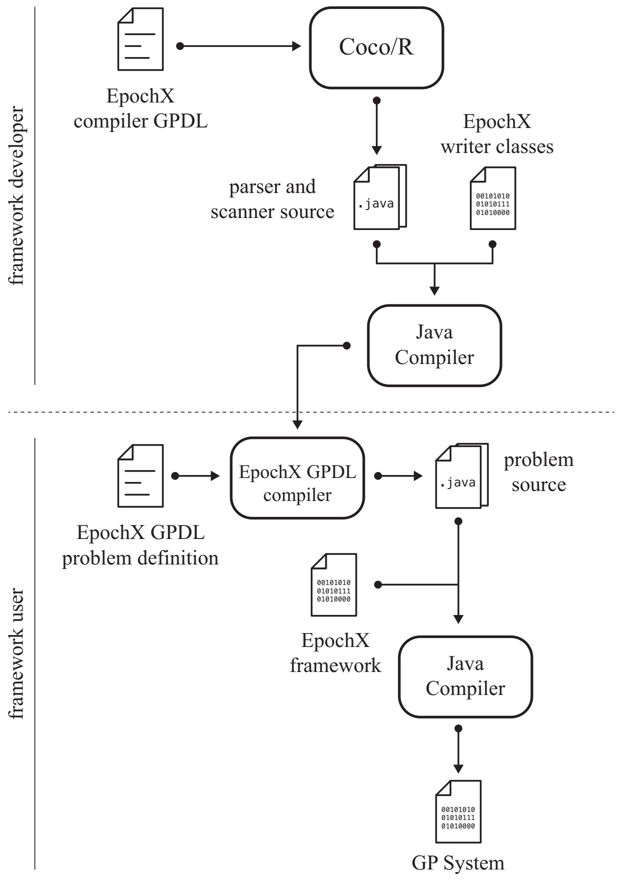

Figure 1: Diagrams showing the steps involved in generating a GPDL compiler (framework developer tasks) and in generating a specific GP system implementation from a GPDL problem description (framework user tasks). Note that the GPDL compiler generation only needs to be done once—it has to be repeated in case the GPDL compiler definition changes.

The scanner and parser created by Coco/R—together with auxiliary file writers discussed in subsection 3.1—are used to create the GPDL compiler for the EpochX framework, which can take a GPDL problem definition file and generate the framework-specific classes required to run the problem. As we mentioned before, while the GPDL language is independent of the target framework, the semantic actions are framework-dependent. EpochX support different representations—context-free grammar (CFG-GP) and grammatical evolution (GE)—and therefore, the same GPDL problem definition file can be reused to generate the source code for

[1]A generic GPDL attribute grammar syntax definition is available at http://dev.heuristiclab.com/GPDL, which can be used as a starting point to create the GPDL definition for the target framework.

these representations. Given the modular architecture of EpochX, there is no need to use a different GPDL definition nor generate different scanner and parser. Depending on the representation chosen, EpochX GPDL parser uses the corresponding representation-specific file writers—these are detailed in subsection 3.1.4.

While EpochX also supports tree-based GP, we have not included support for the tree representation in the GPDL compiler. This is due to the fact that grammar-based representations are a straightforward fit for the GPDL, since the problem definition in GPDL uses grammar rules to define the individual structure. This prevents the problem of not being able to express the rules specified in the grammar or having to employ constraints to the grammar rules—e.g., the GPDL reference implementation in HeuristicLab [8] for tree representation requires that all alternatives in the grammar must contain a single symbol only [2].

3.1 Architecture

3.1.1 Project Manager

The project manager is the most important class in the code generator. It links the parser generated by Coco/R with all other classes involved in generating the problem source code. The current version of EpochX code generator creates an Eclipse[2] project with all required directories and configuration files, as well as the source files.

The main functionality of the project manager is to handle the writing queue, which is created during the parsing of the GPDL problem definition:

```
private ArrayList<WritingInfo> writingQueue;
```

The queue contains the list of files to be generated, including the path of the files and their content. To support multiple representations—CFG-GP or GE in the current implementation—the writers are chosen according to the representation being used.

3.1.2 File Writers

The writer classes are responsible for creating and writing the content of the source files based on the GPDL problem definition. By using separate writers for each section of the GPDL file, it is easy to provide alternative implementations depending on the representation—e.g., while CFG-GP and GE share the same grammar writer, their fitness writer is different given that the individual representation is different.

The different writers and their purpose are:

- `MainWriter:` generates the main class, which provides the facility to run the problem and to set the default parameters. This is a representation-dependent writer;
- `GrammarWriter:` generates the grammar described in the `RULES` section of the GPDL file and sends it to the MainWriter once it is ready;
- `FitnessWriter:` generates the file corresponding to the fitness function, which also contains the code in the `INIT` section of the GPDL file. This is a representation-dependent writer;
- `SymbolWriter:` is a template to define grammar symbols. It contains the information needed for both non-terminal and terminal symbols (i.e., name of the symbol, information concerning its output, list of grammar productions);
- `NonTerminalWriter:` extends the `SymbolWriter` class. It builds the part of the grammar corresponding to each non-terminal symbol and sends it to the GrammarWriter. This writer also produces the source files representing the non-terminal symbols;
- `TerminalWriter:` extends the `SymbolWriter` class. It builds the part of the grammar corresponding to each terminal symbol and sends it to the GrammarWriter. This writer also produces the source files representing the terminal symbols.

It should be noted that the GPDL syntax allows the use of out/ref parameters in symbols of the grammar, which are not supported by Java. To overcome this limitation, `NonTerminalWriter` and `TerminalWriter` use return statements to implement the behaviour of out parameters. There are two uses of out parameters: (i) in the definition of a grammar symbol (left side of the '='); (ii) in the definition of productions (right side of the '='). This is illustrated in the following example:

```
NOT<<Context context, out boolean b>> =
   EXPR<<context, out b>>               SEM<<b = !b;>>
.
```

In the first case (definition of a grammar symbol), the out parameter is used to specify the return type of the method generated to evaluate the symbol. It is also used to define a local variable, which is returned at the end of the method:

```
public Boolean evaluate(Context context) {
   Boolean b;

   // semantic actions

   return b;
}
```

In the second case (definition of productions), the out parameter is used to specify the variable that holds the return value of evaluation of the symbols of the production. In the example above, the use of `out b` specifies that the return of the evaluation of `EXPR` should be stored in the variable `b`:

```
b = (Boolean) get(0).evaluate(context);
```

The method call `get(0)` returns the first symbol of the production (the `EXPR` symbol). Note that at this point the variable `b` already exists, since it is the same variable name as the one in the symbol definition. The complete `NOT` evaluate implementation, including the semantic action, would be:

```
public Boolean evaluate(Context context) {
   Boolean b;

   b = (Boolean) get(0).evaluate(context);
   b = !b;

   return b;
}
```

[2] http://www.eclipse.org/

This example shows two limitations of the current implementation of both `NonTerminalWriter` and `TerminalWriter`. Firstly, it is only possible to specify a single out parameter, since only a single value can be returned. While this can be seen as a limitation, nodes usually evaluate to a single value in most GP systems and therefore, might not present a problem. If there is the need to return multiple values, due to the use of multiple out parameters, the use of struct-like data types could be explored.

Secondly, all symbol implementations follow the contract of the interface `Function`, which specifies the method:

```
public abstract Object evaluate(Context context);
```

Therefore, only the `Context` object can be used as an argument in grammar symbols. This is due to a design decision of re-using the existing EpochX classes to represent the nodes in GP trees. The `Context` class represents the evaluation context as a hash map-like structure, storing pairs of key-value objects, and terminal nodes can query the values of the input variables. Using a fixed interface to evaluate nodes, it is possible to simplify the code of the writers, since the method call to evaluate different symbols is uniform. This does not limit the values that can be passed to symbols—multiple key-value pairs can be set in the same context.

3.1.3 *Error detection*

In addition to the syntax checker provided by the parser, we incorporated additional error checking of the input file. This allows the compiler to check whether the the GPDL problem definition is well-constructed and complies with the requirements of the EpochX framework.

We can distinguish three different types of errors:

- *Grammar definition errors*: when the grammar rules defined in the `RULES` section of the GPDL file is not valid (e.g., it contains duplicated symbols, not declared symbol). Note that a grammar can be syntactically correct and, at the same time, it can represent a malformed grammar;
- *GPDL error*: when the file does not comply with one of the constraints of GPDL. This type of error is originated by the parser;
- *EpochX not compatible file*: when the file does not comply with one of the constraints specific to EpochX (e.g., a symbol can only have one out parameter, an evaluation function must appear in the `MAXIMIZE` or `MINIMIZE` section).

All errors defined above are reported to the parser as semantic errors and detailed with an appropriate error message. The actual position (line and column numbers) is then added by the parser and the message is printed. Since all errors are reported to the parser, we have a global number of both syntactic and semantic errors. At the end of the parsing process of the GPDL file, if the number of errors detected by the parser is zero, the EpochX source code of the problem definition will be generated.

3.1.4 *Representation-specific compiler*

Since EpochX supports the generation of both CFG-GP and GE representations from the same GPDL problem definition, there are two different compilers—one for each representation. At the same time, both compilers share the same parser and most of the file writers, taking advantage of the modular architecture of EpochX. The parser generated by Coco/R is not tied to use a specific file writer, but it uses a reference to the file writer. Therefore, a compiler can set the value of the file writer to be used before invoking the parser.

For example, the CFG-GP and GE use a different `FitnessWriter`. The `ProjectManager` class defines a constant to reference the fitness writer:

```
public static ConfigKey<FitnessWriter> FITNESS_WRITER =
  new ConfigKey<FitnessWriter>();
```

The parser can use this reference to retrieve the fitness writer to be used, without requiring a specific one to be hard-coded:

```
Config.getInstance().get(ProjectManager.FITNESS_WRITER)
```

Each compiler can then set its corresponding fitness writer before invoking the parser:

```
// sets the GE fitness writer
Config.getInstance().set(ProjectManager.FITNESS_WRITER,
  new FitnessWriterGE());
```

Given that both CFG-GP and GE are grammar-based representation, there is no need to provide specific writers for the grammar, symbols, non-terminals and terminals. The only other representation-specific writer is the `MainWriter`, which is responsible to write the file that contains the default parameters values as well as to set the genetic operators.

One advantage of being able to generate the code to different GP representations from the same GPDL problem definition is that a user can easily compare the results using different GP representations, while just having to write a single GPDL file. There is no need to manually write representation-specific classes, significantly reducing (or in some cases even eliminating) the effort of experimenting with alternative representations.

4. EXAMPLE OF AN EPOCHX GPDL PROBLEM DEFINITION

The following listing presents a GPDL problem definition for the N-Multiplexer problem compatible with EpochX. The start of the file contains the problem name:

```
PROBLEM Multiplexer
```

The `CODE` section specifies the number of variables dynamically, which will be used to define the terminal symbols of the problem. The size of the Multiplexer problem is specified by the `address` variable, which correspond to the number of address inputs available:

```
CODE<<
  import org.epochx.gpdl.tools.Tools;

  private int address = 3;
  private int size = (int)Math.pow(2,address) + address;
  private boolean[][] data =
    new boolean[(int) Math.pow(2,size)][];

  public String[] variables = Tools.getVariables(
    new Tools.Variable[] {
      new Tools.Variable("a", address),
      new Tools.Variable("d", size - address)});
>>
```

The data array representing the fitness cases is initialised once at the beginning of the run:

```
INIT<<
  import java.math.BigInteger;

  for (int i = 0; i < data.length; i++) {
    data[i] = new boolean[size];
    int mask = BigInteger.ONE.shiftLeft(size - 1)
      .intValue();

    for (int j = 0; j < data[i].length; j++) {
      data[i][j] = (i & (new BigInteger(
        String.valueOf(mask)).shiftRight(j).intValue()))
          > 0;
    }
  }
>>
```

Each non-terminal symbol is then declared:

```
NONTERMINALS
  PROGRAM<<Context context, out boolean b>>.
  EXPR<<Context context, out boolean b>>.
  AND<<Context context, out boolean b>>.
  OR<<Context context, out boolean b>>.
  IF<<Context context, out boolean b>>.
  NOT<<Context context, out boolean b>>.
```

The terminals are defined using the names from the `variables` using the GPDL constraints facility:

```
TERMINALS
  VAR<<Context context, out String value>>
    CONSTRAINTS
    value IN SET <<variables>> .
```

The productions of the previously declared non-terminal symbols are now defined in the `RULES` section, which represents the grammar. Note that the semantic action of each non-terminal is also defined here, following the GPDL syntax:

```
RULES

  PROGRAM<<Context context, out boolean b>> =
    EXPR<<out b>>
  .

  EXPR<<Context context, out boolean b>> =
                               LOCAL<<String value;>>
    AND<<context, out b>>
    | OR<<context, out b>>
    | IF<<context, out b>>
    | NOT<<context, out b>>
    | VAR<<context, out value>>
              SEM<<b = (boolean) context.get(value);>>
  .

  AND<<Context context, out boolean b>> =
                              LOCAL<<boolean b1,b2;>>
    EXPR<<context, out b1>> EXPR<<context, out b2>>
                                  SEM<<b=b1 && b2;>>
  .

  OR<<Context context, out boolean b>> =
                              LOCAL<<boolean b1,b2;>>
    EXPR<<context, out b1>> EXPR<<context, out b2>>
                                  SEM<<b=b1 || b2;>>
  .

  NOT<<Context context, out boolean b>> =
    EXPR<<context, out b>>                 SEM<<b=!b;>>
  .

  IF<<Context context, out boolean b>> =
                                LOCAL<<boolean test;>>
    EXPR<<context, out test>>         SEM<<if (test){>>
    EXPR<<context, out b>>              SEM<<} else {>>
    EXPR<<context, out b>>                     SEM<<}>>
  .
```

Finally, the last part of the problem definition specifies the fitness function. EpochX requires that fitness functions have the same method signature:

```
public double evaluate(Function executable,
                       Context context)
```

The first argument is the program (expression) represented by the current individual being evaluated; the second argument is the evaluation context object. While we could have omitted the requirement of having to specify the method signature, we decided to keep it to make explicit the objects available for the fitness function.

Note `Context` object is used to set the input values of the current fitness to the variables—these values can be looked up using variable name. At the end of the file, `END` section closes problem the definition:

```
MINIMIZE<<
  public double evaluate(Function executable,
                         Context context) {
    int correct=0;

    for (int i=0; i<data.length ; i++) {
      int expected = 0;
      int index = data[i].length - 1;

      for (int j = index; j >= 0; j--) {
        if (j < address) {
          context.set("a" + ((address - 1) - j),
                      data[i][j]);

          if (data[i][j]) {
            expected += Math.pow(2, (address - 1) - j);
          }
        } else {
          context.set("r" + (index - j), data[i][j]);
        }
      }

      boolean result = (boolean) executable
        .evaluate(context);

      if (data[i][index - expected] == result) {
        correct++;
      }
    }

    return data.length - correct;
  }
>>

END Multiplexer.
```

5. CONCLUSION

In this paper we presented a GPDL code generator for the EpochX framework using a GPDL-enabled compiler based on a parser generated by Coco/R. The aim of the compiler is to generate the source code of the corresponding GP system structured as an Eclipse project, given a GPDL problem definition. We discussed the steps involved in the compiler generation, which only need to be performed once and they should be repeated in case of a new GPDL syntax specification. We identified an interesting advantage of using GPDL as a problem definition: it allows a user to write a single problem definition to generate the corresponding source code for different representations (e.g., CFG-GP and GE in the current implementation), facilitating the experimentation of different representations.

There are several improvements that can be made to the current implementation. At the moment, the default parameters (e.g., population size, genetic operators probabilities) are not specified in the GPDL problem definition, nor are the selection of genetic operators. We are currently exploring alternatives to relax the requirement that all symbols implement the interface `Function`, to omit the fitness function signature from the GPDL problem definition and we are also investigating the implementation of the tree-based GP compiler. The current version of the compiler generates the source code and places them in a Eclipse project structure. Nothing prevents the EpochX compiler to call the Java compiler immediately after the generation of the files to start the execution of the algorithm, as it occurs in the HeuristicLab reference implementation [2]—this is something that can also be explored. The requirement of having language-specific code and text in the problem definition file is not ideal. This could be improved by providing a GPDL editor, which could include an inline syntax analyser. Another possibility is to separate the textual definitions and the language-specific code in different files, facilitating the reuse of the GPDL definition across different frameworks.

EpochX is available for download, including source code and documentation at: `http://www.epochx.org/`

6. ACKNOWLEDGEMENTS

Claris Leroux thanks the ERASMUS *Student Mobility for Placements*, the *Conseil Général de Haute Normandie* and the *Ministère de l'Éducation Nationale (Aide à la Mobilité Internationale)* for their scholarships.

7. REFERENCES

[1] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[2] G. Kronberger, M. Kommenda, S. Wagner, and H. Dobler. Gpdl: A framework-independent problem definition language for grammar-guided genetic programming. In *GECCO'13 Companion*, pages 1333–1340. ACM, 2013.

[3] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[4] H. Mössenböck. A generator for fast compiler front-ends. In *Report 127, Dept. Informatik, 28 pages.* ETH Zürich, 1990.

[5] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, Aug. 2001.

[6] F. Otero, T. Castle, and C. G. Johnson. Epochx: Genetic programming in java with statistics and event monitoring. In *GECCO'12 Companion*, Philadelphia, PA, USA, July 2012.

[7] L. Vaseux, F. Otero, T. Castle, and C. G. Johnson. Event-based graphical monitoring in the epochx genetic programming framework. In *GECCO'13 Companion*, Amsterdam, The Netherlands, July 2013.

[8] S. Wagner. *Heuristic Optimization Software Systems – Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment.* PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria, 2009.

[9] P. Whigham. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, 1995.