GPDL: A Framework-Independent Problem Definition Language for Grammar-Guided Genetic Programming

Gabriel Kronberger
gabriel.kronberger@fh-hagenberg.at

Michael Kommenda
michael.kommenda@fh-hagenberg.at

Stefan Wagner
stefan.wagner@fh-hagenberg.at

Heinz Dobler
heinz.dobler@fh-hagenberg.at

University of Applied Sciences Upper Austria
School for Informatics, Communications and Media
Softwarepark 11, 4232 Hagenberg, Austria

ABSTRACT

Defining custom problem types in genetic programming (GP) software systems is a tedious task that usually involves the implementation of custom classes and methods including framework-specific code. Users who want to solve a custom problem have to know the details of the targeted framework, for instance cloning semantics, and often have to write a lot of boilerplate code in order to implement the necessary functionality correctly. This can lead to frustration and hinders new developments and the application of GP to solve interesting problems.

In this contribution we propose a framework-independent definition language for GP problems that can reduce the required effort and facilitate the integration of new problem types. We draw a parallel between the implementation of compilers for programming languages and the implementation of GP problems and reuse the well-established concept of attributed grammars with semantic actions to define computational symbols, semantics and structural constraints for GP. This goes beyond previous work in the area of context-free-grammar GP and grammatical evolution, because we also interweave the definition of symbol semantics and the target function with the definition of the grammar.

This paper describes the proposed GP problem definition language (GPDL) and exemplary definitions of two popular benchmark problems using GPDL. We also describe a reference implementation of a GPDL compiler for HeuristicLab.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program synthesis; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic Methods

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

Keywords

Domain Specific Languages, Evolutionary Computation Software Systems, Genetic Programming

1. MOTIVATION

Most genetic programming software systems are not simple enough to be readily used as standard tools for solving optimization problems. In many GP systems (e.g., HeuristicLab, ECJ, JGAP, and OpenBEAGLE) the definition of a custom GP problem usually includes the implementation of a number of classes, including classes for symbols, classes for constraints and classes for evaluation. In this process, users must also write a lot of framework-specific boilerplate code, e.g., for cloning, persistence, or algorithm analysis. As a result, a large part of the source code is not directly related to the problem definition and is only necessary to fit the classes into the existing framework. Users therefore have to know implementation details to be able to implement the functionality correctly.

As a result, the necessary effort for implementing even simple problems (e.g., symbolic regression or artificial ant) is comparatively large and can be estimated at several days for inexperienced users. The effort for larger and more complex problems is often much higher, especially if the symbol semantics or structural constraints are complex. This is too much effort for users who are primarily interested in solving a given problem using GP, or only experimenting if GP would be a usable method at all. As a result, GP systems are not used frequently, because the effort is prohibitively large.

Another important aspect is that several sophisticated and powerful GP systems are available, but a user would have to implement a problem for each framework separately in order to try which system is most suitable.

Our suggestion to improve the current situation is to define a domain specific language (DSL) for the definition of GP problems. This language should be easy to understand, and independent of the framework as well as the programming platform. Using this language, it should be possible to define GP problems in an abstract way so that it is possible to solve these problems with various popular GP software systems.

A similar development has occurred in the closely related area of mathematical programming or constraint programming. A large number of solvers using sophisticated algorithmic machinery have been developed and partially canned into software products (e.g., CPLEX solver). Domain specific languages for mathematical programming (e.g., OPL or AMPL) have been developed to allow the formulation of such optimization problems independently from specific solvers. An important aspect of such languages is the separation of the problem formulation from the solver algorithm. Many solvers support standard languages so that it is easy to experiment with different solvers.

Generally, GP can be seen as a solver for problems where the goal is to find a solution in form of a program which is optimal with respect to the given target function. There are many different approaches to GP which all fit into this general description, and also other kinds of solvers which are applicable to this problem.

In this contribution we describe the first implementation of a framework-independent language for GP problem definitions (GPDL). We aim to raise awareness of the issues discussed above and to facilitate discussion on the viability and usefulness of the proposed solution among the community of GP software system developers. We also describe the reference implementation of a compiler for GPDL targeting the HeuristicLab framework [11],[6]. The reference implementation should help developers who want to implement the language for other GP software systems.

2. RELATED WORK

Context-free grammars (CFG) have already been used to define the set of possible solution candidates in context-free-grammar GP [13] and grammatical evolution [9]. This is a powerful idea that also seems natural because the promise of GP is to find programs solving a given problem and programs are usually specified in a programming language with syntax defined by a grammar. A recent survey of grammar guided GP approaches is given in [8].

Some popular GP paradigms, for instance stack-based GP systems, most notably PushGP, or other forms of linear GP that evolve assembler, binary or byte code directly, use a very simple syntax. These systems rely more on the semantics of operations; the set of computational symbols is predefined and fixed, so the evolutionary operators, crossover and mutation, are designed to take operation semantics into account. In these GP systems the set of possible solution candidates is also defined by a grammar but the semantics are much more important, so the grammar-based approach for problem definition does not fit very well. Similarly, Cartesian GP is also not well-suited as it uses a graph-based representation for solution candidates.

Software systems implementing CFG-GP or grammatical evolution (e.g., EpochX, GEVA) already allow specifying the symbol set and the grammar using a BNF-like notation. However, the symbol semantics have to be implemented separately in form of an evaluation function that receives the produced solution candidate as input and calculates its fitness value. This has to be implemented as a separate Java class that integrates into the framework.

Compiler generators are standard tools for the implementation of parsers, interpreters or compilers for programming languages. Based on a language specification, a compiler generator produces a parser or part of a compiler for this language. The language specification is usually based on an attributed grammar [5] with semantic actions. The grammar defines the syntax and the language semantics are defined through a combination of semantic actions and symbol attributes. Typically, semantic actions are specified using source code in the target programming language. The compiler generator interweaves the source code for semantic actions with the generated source code for parsing the input token stream.

In the definite clause translation grammar GP (DCTG-GP) [10] system the concept of attributed grammars with semantics actions was already used for the definition of GP problems to reduce the implementation effort. As in other CFG-GP or GE systems, the problem description encompasses the set of computational symbols and the grammar for solution candidates. However, the definition of symbol semantics is also included in the problem description document. Including semantic actions directly into the problem definition has the benefit that it is possible to specify all details of the GP problem in one self-contained document without writing boilerplate code. McKay et al. state that definition of semantics in DCTG-GP *"is substantially simpler than for a standard GP system"*, and that *"the programming cost of targeting a new problem is generally small – in the case of DCTP-GP, quite typically ten or twenty lines of code"* [8].

However, the DCTG-GP problem definition language is rather hard to read and specific to Prolog; additionally, the fitness function has to be implemented separately. The language format has not been used in any other GP systems besides DCTG-GP.

3. DESIGN CONSIDERATIONS

Based on our research of previous work and the initial motivation of our idea, our design goals for GPDL are simplicity, expressivity, and generality:

- Simplicity: Users should be able to define custom problems in less time, so that it is possible to quickly try if the problem can be solved by GP and to try different variants of the problem formulation. In particular, it should not be necessary to familiarize oneself with the details of the target framework and it should not be necessary to write boilerplate code to integrate a new problem. Furthermore, it should not be necessary to separately recompile code when adapting or extending the problem definition. In an exaggerated scenario, it should be possible to simply call a "solver" executable with a GP problem definition so that the solver produces a solution and its fitness value after a specified time interval. GPDL should have a simple, easily understandable syntax so that problem definitions are easy to read.

- Expressivity: The language should allow specification of problems in a way that they can subsequently be solved by a large number of GP software implementations. Many GP implementations support a form of grammar-guided GP either using a tree-based representation and structural constraints (e.g., for strongly-typed GP) or via grammatical evolution. It should be possible to formulate the commonly used GP benchmark problems in a way that makes it possible to solve such problems. Of course, the language should not be

limited to the restricted set of problems, where the symbols set fulfills the closure property.

- Generality: The problem definition should be independent of the target framework and of the search algorithm or GP paradigm. GP systems share many commonalities, especially regarding the representation of solution candidates, which are frequently either directly encoded using a tree data structure or mapped from a linear representation to a tree using a grammar. The differences lie mainly in the details of the search procedure, e.g., the pipeline of the evolutionary search or the used operators. Ideally, it should be possible to attack the problem using different solvers and search paradigms (e.g., evolutionary and tree based, grammatical evolution, estimation of distribution and probabilistic context free grammars [4]) using the same definition file. Users are often not primarily interested in the details of the algorithm used to find a solution.

GP variants such as PushGP that use a linear representation or variants operating directly on machine code or byte code are not well suited for the proposed problem definition languages, because in these languages the set of computational symbols and their semantics are usually pre-defined and fixed and because they have very simple syntax. Cartesian GP, which uses a graph representation, is another notable exception which is not well suited for the proposed problem definition language.

4. DEFINITION OF GPDL

In the following we describe GPDL in detail. It should be noted that even though GPDL is similar to languages for compiler construction, its purpose is different. A problem definition in GPDL is used to generate and interpret sentences using the specified grammar. This is in contrast to languages for compiler construction which are used to parse and interpret existing input sentences. In particular, a solver (e.g., genetic programming) uses a GPDL definition to generate solution candidates and search for optimal solutions.

For the reference implementation of GPDL we use the Coco-2 compiler generator [3],[2] to produce the GPDL compiler for HeuristicLab. Thus, in the following we use the Coco-2 compiler definition language for the specification of GPDL. Coco-2 is available for multiple platforms including C#, Java and C++, has a descriptive meaningful syntax, and allows to specify the grammar using EBNF-notation[1]. Figure 1 shows the definition of a symbolic regression problem in GPDL. The syntax definition of GPDL itself is shown on the right; this specification can be used to generate a GPDL syntax analyzer with Coco-2.

4.1 Syntax

The syntax of GPDL is very similar to the syntax of the Coco-2 compiler definition language [3] because both languages allow definition of an attributed grammar in EBNF syntax with semantic actions. A GP problem definition has a name and consists of four basic parts: the set of non-terminal symbols (`NONTERMINALS`), the set of terminal symbols (`TERMINALS`), the set of rules (`RULES`), and a target function (`MAXIMIZE` or `MINIMIZE`). Additionally, there can be an optional part including any kind of additional source code (`CODE`) and source code for initialization (`INIT`).

```
COMPILER GPDef
CHARACTER SETS
  letter = 'A'..'Z' + 'a'..'z'.
  digit = '0'..'9'.
  whiteSpace = CHR(9) + EOL IGNORE.
COMMENTS FROM '/*' TO '*/' NESTED.
KEYWORDS
  'PROBLEM'. 'END'. 'EPS'. 'LOCAL'. 'SEM'.
  'NONTERMINALS'. 'TERMINALS'. 'RULES'.
  'MAXIMIZE'. 'MINIMIZE'. 'INIT'. 'CODE'.
  'CONSTRAINTS'. 'IN'. 'SET'. 'RANGE'.
TOKENS
  '='. '|'. '.'. '('. ')'. '['. ']'.
  '{'. '}'. '<<'. '>>'. '..'.
TOKEN CLASSES
  ident = letter {letter | digit}   .
NONTERMINALS
  GPDef. NTDecl. TDecl.
  RuleDef. SynExpr. SynTerm. SynFact. SemAction.
  FormAttrList. ActAttrList. SrcText. ConstrDef.
RULES
  GPDef        = 'PROBLEM' ident
                 ['CODE' SrcText ]
                 ['INIT' SrcText ]
                 'NONTERMINALS' { NTDecl }
                 'TERMINALS' { TDecl }
                 'RULES' { RuleDef }
                 ('MAXIMIZE' | 'MINIMIZE') SrcText
                 'END' ident '.' .
  NTDecl       = ident FormAttrList '.' .
  TDecl        = ident FormAttrList
                 'CONSTRAINTS' { ConstrDef } '.' .
  ConstrDef    = ident 'IN' ('SET' SrcText
                 | 'RANGE' SrcText '..' SrcText ).
  RuleDef      = ident FormAttrList '='
                 ['LOCAL' SrcText ] SynExpr '.' .
  SynExpr      = SynTerm { '|' SynTerm } .
  SynTerm      = SynFact { SynFact } .
  SynFact      = ident ActAttrList
                 | 'EPS'
                 | SemAction | '(' SynExpr ')'
                 | '[' SynExpr ']'
                 | '{' SynExpr '}' .
  SemAction    = 'SEM' SrcText .
  SrcText      = '<<' /* code */ '>>' .
  FormAttrList = '<<' /* formal param. */ '>>' .
  ActAttrList  = '<<' /* actual param. */ '>>' .
END GPDef.
```

Non-terminal symbols (`NTDecl`) can only occur as inner nodes and the root node of a parse tree and there must be exactly one matching rule in the `RULES` section for each non-terminal symbol. Terminal symbols (`TDecl`) contain data components and occur only as leaves of the parse tree. The allowed values for the data components of terminal symbols can be defined in the `CONSTRAINTS` section using either a set of allowed values or a range of values (`SET, RANGE`).

Rules (`SynExpr`) can be specified using the symbols of EBNF-notation and can be interspersed with semantic actions (`SEM`). Only context-free grammars are supported so each rule consists of a set of alternative productions for each non-terminal symbol.

GPDL is independent of the programming language of the target framework. Semantic actions can be defined using source code fragments between the `<<` and `>>` tags in the three non-terminal symbols (`SrcText`, `FormAttrList`, and `ActAttrList`). These source code fragments are directly copied to the produced output code and can use any

[1]The closely related variant Coco/R is available under the GPL from `http://www.ssw.uni-linz.ac.at/coco/`

```
PROBLEM SymbRegKoza
CODE <<
  double[,] inputValues;  double[] targetValues;  string[] variableNames;
  double GetValue(double[,] data, string varName, int row) { /* ... */ }
  double RSquared(IEnumerable<double> xs, IEnumerable<double> ys) { /* ... */ }
  void LoadData(out double[,] inputValues, out string[] variableNames, out double[] target) { /* ... */ }
>>
INIT <<
  LoadData(out inputValues, out variableNames, out targetValues);
>>
NONTERMINALS
  Model<<int row, out double val>>. RPB<<int row, out double val>>.
  Addition<<int row, out double val>>. Subtraction<<int row, out double val>>.
  Multiplication<<int row, out double val>>. Division<<int row, out double val>>.
TERMINALS
  ERC<<out double val>>
    CONSTRAINTS
      val IN RANGE <<-100>> .. <<100>>
  .
  Var<<out string varName>>
    CONSTRAINTS
      varName IN SET <<variableNames>>
  .
RULES
  Model<<int row, out double val>> =
    RPB<<row, out val>>
  .
  RPB<<int row, out double val>> =              LOCAL << string varName; >>
    Addition<<row, out val>>
    | Subtraction<<row, out val>>
    | Division<<row, out val>>
    | Multiplication<<row, out val>>
    | Var<<out varName>>                        SEM << val = GetValue(inputValues, varName, row); >>
    | ERC<<out val>>
  .
  Addition<<int row, out double val>> =         LOCAL << double x1, x2; >>
      RPB<<row, out x1>> RPB<<row, out x2>>     SEM<< val = x1 + x2; >>
  .
  Subtraction<<int row, out double val>> =      LOCAL << double x1, x2; >>
      RPB<<row, out x1>> RPB<<row, out x2>>     SEM<< val = x1 - x2; >>
  .
  Division<<int row, out double val>> =         LOCAL << double x1, x2; >>
      RPB<<row, out x1>> RPB<<row, out x2>>     SEM<< val = x1 / x2; >>
  .
  Multiplication<<int row, out double val>> =   LOCAL << double x1, x2; >>
      RPB<<row, out x1>> RPB<<row, out x2>>     SEM<< val = x1 * x2; >>
  .
MAXIMIZE  <<
    var rows = System.Linq.Enumerable.Range(0, inputValues.GetLength(0));
    var predicted = rows.Select(r => {
      double result;
      Model(r, out result);                     /* we can call the root symbol directly */
      return result;
    });
    return RSquared(predicted, targetValues);   /* target function returns double */
  >>
END SymbRegKoza.
```

Figure 1: Exemplary definition of a simple symbolic regression problem in GPDL.

constructs available in the targeted programming language. The problem is specified in a declarative way which does not specify details of the search procedure, thus, GPDL is independent of the search or optimization algorithm.

4.2 Semantics

A compiler for a tree-based GP framework must typically produce classes for all symbols, code for checking constraints, a class for the target function and code to evaluate or interpret solution candidates. Often the code for interpretation is either included directly in the symbols or in the class for the fitness evaluation. A GPDL document contains all necessary information to produce the required classes. The set of symbols can be derived directly from the `TERMINALS` and `NONTERMINALS` section. The fitness function can be extracted directly from the `MAXIMIZE` or `MINIMIZE` section. Finally, the `RULES` section contains information that is relevant both for the symbol constrains and also for the interpretation of symbols (`<<..>>`).

4.2.1 Symbols

In tree-based GP the set of non-terminal symbols and the set of terminal symbols directly relate to the function set and the terminal set, respectively. It seems natural to think of non-terminal symbols as functions or operators taking arguments and terminal symbols as literal values. However, it should be noted that this is not a strict relation in general, as any kind of semantics is valid for terminal and non-terminal symbols in GPDL.

The major difference between terminal symbols and non-terminal symbols is that terminal symbols can be composed of multiple data components. All "out"-parameters of the formal parameter list of terminal symbols are considered as local data components that can be different for each occurrence of the symbol in solution candidates. This is necessary, e.g., for the implementation of ephemeral random constants (ERC) which are initialized on tree generation (compare Figure 1). The allowed values for each data component have to be specified in the `CONSTRAINTS` section. For categorical variables the set of allowed values can be specified directly using the `SET` keyword and an expression returning the allowed values (i.e. `IEnumerable<T>` in C#). For continuous variables, a range of values can be specified using the `RANGE` keyword and two expressions returning the minimum and maximum values. Non-terminal symbols cannot contain data components, but as a work-around it would be possible to define the grammar in a way, that such a non-terminal symbol always has a terminal child symbol containing the data components.

4.2.2 Grammar

The translation of grammar rules is very specific to the capabilities of the target GP system. Therefore, the details of the grammar transformation are not specified in this contribution. In grammatical evolution systems it is necessary to transform the rule definitions to match the rule definitions of the target system. In GP systems supporting tree-based GP with symbol constraints, the rules and alternatives have to be transformed to symbol constraints. If the constraint system of the target framework is not powerful enough to express the rules specified in the grammar, then either the compiler is able to transform the grammar automatically or the user has to adapt the grammar accordingly.

The reference implementation for HeuristicLab described below only supports grammars that do not contain repetitions `{..}` and/or optional chains `[..]`. Additionally, all alternatives must contain a single symbol only.

4.2.3 Interpretation

For the interpretation of solution candidates, the production rules have to be translated into source code for evaluation. This can be done in a recursive-descent fashion where an evaluation method is generated for each symbol. The production rules can then be translated into source code containing calls to other evaluation methods and the source code specified in semantic actions can be integrated at the correct positions directly.

Both, non-terminal and terminal symbols, can have attributes which can be used either to supply additional data that is necessary for evaluation from outside, or to produce output values. Through attributes it is possible to define multiple output values of different types. Formal attributes are translated by the GPDL compiler to formal parameters for each non-terminal symbol. In production rules, actual attribute values have to be specified if a symbol with attributes is used on the right-hand-side. The actual attribute values are translated by the GPDL compiler to actual parameters values for the invocation of the evaluation method.

4.2.4 Scope

Variables and methods defined in the source code in the `CODE` section can be used from all sections that allow specification of source code. The statements specified in the `INIT` section are executed once when initializing the problem class (i.e. called from the constructor of the generated problem class). Variables defined in the `LOCAL` section for rules or in semantic actions are visible only within the rule. Variables defined in the `MAXIMIZE/MINIMIZE` section are only visible within this section.

4.3 Examples

Figure 1 shows the GPDL definition of the familiar symbolic regression problem. This is a simple example because the symbol set fulfills the closure property [7] and functions are side-effect free. The arithmetic operators represented as separate non-terminal symbols each take two arguments and produce a single output value. Two types of terminal symbols represent ephemeral random constants (`ERC`) and variables `Var`. The `ERC` symbol contains the data component `val` to store the constant value which is initialized randomly in the range $[-100..100]$ and is constant over the whole run. The `Var` symbol contains the name of the variable it represents. This is also initialized by randomly selecting an element from the set of available variable names. The target function is the sum of squared errors which should be minimized. On the first call of the target function, the data is loaded from a file. The root symbol of the grammar (`Model`) serves as the entry point for the evaluation of a solution candidate in the target function. For each row in the data set, the method for the root symbol is called specifying the `row` value as an input. The result of the evaluation is returned via the output parameter `val`.

Note that we left out some less relevant parts in both examples due to page constraints. The full source code for these examples is available on our website[2].

[2] `http://dev.heuristiclab.com/GPDL`

```
PROBLEM MultiOutputMultiplier
CODE <<
  const int N = 4;
  class State { public bool[] x1; public bool[] x2; public bool[] output; }
  void GenerateProblemData(int n, out bool[][] a, out bool[][] b, out bool[][] expectedOutput) {
    /* ... */
  }
>>
NONTERMINALS
  Expr<<State state, out bool res>>.
  Assign<<State state, out bool res>>.
  AND<<State state, out bool res>>.
  AND1<<State state, out bool res>>.
  XOR<<State state, out bool res>>.
  OR<<State state, out bool res>>.
TERMINALS
  InputA<<out int id>>
    CONSTRAINTS
      id IN SET << Enumerable.Range(0, N) >> .
  InputB<<out int id>>
    CONSTRAINTS
      id IN SET << Enumerable.Range(0, N) >> .
  Output<<out int id>>
    CONSTRAINTS
      id IN SET << Enumerable.Range(0, 2 * N) >> .
RULES
  Expr<<State state, out bool res>> =                LOCAL << int id; >>
    Assign<<state, out res>>
    | AND<<state, out res>>
    | AND1<<state, out res>>
    | XOR<<state, out res>>
    | OR<<state, out res>>
    | InputA<<out id>>                               SEM<< res = state.x1[id]; >>
    | InputB<<out id>>                               SEM<< res = state.x2[id]; >>
    | Output<<out id>>                               SEM<< res = state.output[id]; >>
  .
  Assign<<State state, out bool res>> =              LOCAL << int outId; >>
    Output<<out outId>>  Expr<<state, out res>>      SEM<< state.output[outId] = res; >>
  .
  AND<<State state, out bool res>> =                 LOCAL << bool resA, resB; >>
    Expr<<state, out resA>> Expr<<state, out resB>>  SEM << res = resA & resB; >>
  .
  AND1<<State state, out bool res>> =                LOCAL << bool resA, resB; >>
    Expr<<state, out resA>> Expr<<state, out resB>>  SEM << res = resA & !resB; >>
  .
  XOR<<State state, out bool res>> =                 LOCAL << bool resA, resB; >>
    Expr<<state, out resA>> Expr<<state, out resB>>  SEM << res = resA ^ !resB; >>
  .
  OR<<State state, out bool res>> =                  LOCAL << bool resA, resB; >>
    Expr<<state, out resA>> Expr<<state, out resB>>  SEM << res = resA | resB; >>
  .
MINIMIZE <<
    bool[][] a, b, expectedOutput;
    GenerateProblemData(N, out a, out b, out expectedOutput);
    int sumErr = 0;
    for(int i=0; i<expectedOutput.Length; i++) {
      bool tmp;
      // evaluate tree for inputs a and b
      var state = new State();
      state.x1 = a[i]; state.x2 = b[i]; state.output = new bool[2*N];
      Expr(state, out tmp);

      // count incorrect bits
      for(int j=0;j<state.output.Length;j++) {
        if(state.output[j] != expectedOutput[i][j]) sumErr++;
      }
    }
    return (double)sumErr;
  >>
END MultiOutputMultiplier.
```

Figure 2: Exemplary definition of a 4-bit multi-output multiplier problem [12] in GPDL.

Figure 2 shows a GPDL definition of the multi-output multiplied problem [12]. This is an example for a more complicated problem which is not straight-forward to implement in a simple tree-based GP system. This problem has originally been formulated for a Cartesian GP system which naturally supports multiple output values. In our adaptation for grammar guided GP systems, a solution candidate is a single boolean expression that consists of operator symbols and terminal symbols. We represent input lines and output lines using terminal symbols and introduce an assignment operator that can be used to set output values. Output lines can again be used as inputs in expressions. The assignment operator also returns the assigned right hand side value so the operator can again be used as part of an expression. The semantics of Boolean operators (`AND, XOR, OR`) are defined as expected; the symbol `AND1` represents a logical AND with one input negated. Each operator produces a single output bit from two input bits and the terminal symbols produce a single bit as a result. However, the symbols also have attributes for the full input and output vectors, so it is possible to set selected output bits via the assignment operator. The target function returns the number of incorrect output bits produced by a solution candidate for two input vectors.

On the GPDL website[2] we provide additional examples.

5. LIMITATIONS

The target framework must have support to restrict solution candidates. For instance, if the target framework supports tree-based GP but only allows specification of the number of children of function symbols, then it would not be possible to map grammar rules to the less powerful set of constraints. In such cases, only a subset of the allowed specifications can be translated to the target framework. Ideally, if the target framework already uses context free grammars for solution candidates only minor adaptations and checks (e.g., the LL(1) property) are necessary. Software systems that support tree-based GP with structural and symbol constraints (e.g., HeuristicLab, ECJ, JGAP, OpenBEAGLE, ...) and frameworks that support CFG-GP, strongly-typed GP, or grammatical evolution (EpochX, GEVA, DEAP, ...) are candidates that could be used as target frameworks because in these systems the necessary functionality to incorporate grammatical rules is already available.

GP systems such as Cartesian GP, PushGP and other linear GP variants already use pre-defined languages with simple syntax, thus these systems are not well suited to the proposed approach and are not considered as solvers for problems specified in GPDL. However, it would be possible to define the stack-based PushGP language in GPDL to emulate stack-based execution of PushGP programs using other GP systems.

6. REFERENCE IMPLEMENTATION

Our implementation consists of a set of C# source files and the attributed grammar for GPDL from which we generate additional C# source files using Coco-2. From the resulting files we build the compiler for HeuristicLab. The compiler consists of three main parts: the parser (generated by Coco-2), the code generator the compiler and executer. The parser reads a GPDL problem definition as shown in the examples section and builds an abstract syntax tree (AST) data structure in memory. The AST is then supplied to the code generator which uses a number of template files containing all boilerplate code and the information stored in the AST to produce C# source code defining a set of classes for HeuristicLab. In this process, the code generator generates a symbol class for each symbol, a node class for each terminal symbol, a grammar class, an evaluator class and an problem class. Finally, the GPDL compiler invokes the CodeDom C# compiler to compile the generated source files and invoke a genetic programming algorithm configured to solve the custom problem. Details on the implementation of the GP framework in HeuristicLab are given in [6].

6.1 Symbols

The set of non-terminal and terminal symbols can be easily derived from the AST. For terminal symbols it is necessary to generate specific tree node classes to store the data components. The code for the initialization and mutation of terminal symbols is generated based on the specified `CONSTRAINTS`.

6.2 Grammar

The reference implementation does not support all possible problem definitions because HeuristicLab only has limited support for structural constraints for tree-based GP. It is possible to restrict the number of sub-trees and the set of allowed symbols for each sub-tree slot for each symbol. For instance, it is possible to define that an *If-Then-Else* symbol must have exactly three arguments where the first argument can be any symbol from the set of Boolean functions and terminals and the second and third arguments can be any symbol from the set of real-valued functions and terminals. The production rule $A \rightarrow BC|DE$ cannot be translated because the first symbol limits the set of allowed symbols for the second argument. However, note that this rule can be transformed into three rules by introducing two new non-terminal symbols $A \rightarrow F|G$, $F \rightarrow BC$ and $G \rightarrow DE$. which could be modeled using the available set of structural constraints in HeuristicLab.

6.3 Interpreter

The compiler translates symbols and rules using a recursive-descent approach. This means that for each symbol the compiler creates a method that evaluates other symbols in the order stated in the rules section. The methods for interpretation are defined in the problem class which contains all other source code fragments. The compiler must check if the specified grammar fulfills the LL(1) criterion so that an interpreter can be generated using the recursive-descent approach.

One critical aspect that needs to be addressed is how the stream of symbols and the look-ahead symbol are produced. Parsers usually receive a stream of tokens produced by a lexer (lexical scanner). The lexer produces tokens from an input stream (i.e., a text file). In our reference implementation for tree-based GP it is also necessary to process symbols in a sequence and a look-ahead symbol is necessary to choose the correct alternative in the interpreter. In tree-based GP, the input for the interpreter is a tree instead of a stream of characters so a lexer is not necessary. Instead, the stream of symbols is produced by iterating the tree in pre-order.

7. OPEN TOPICS

First and foremost we have not yet analyzed or discussed how problems defined in GPDL can be solved efficiently. The language rules for solution candidates are strongly related to the search over possible solution candidates. Thus, language rules introduce a search bias. Often, multiple different formulations of the same problem are possible which are not equally suited to a given search algorithm. Some formulations of the problem might lead to better search performance than others. The search bias introduced by properties of the language needs to be researched in more detail. Currently, we can only refer to previous work and success stories in the area of grammatical evolution demonstrating the viability of the approach [9].

The current draft specification does not support multi-objective genetic programming. If multi-objective GP should be supported, the specification must be adapted accordingly for instance by allowing multiple definitions of target functions using the `MAXIMIZE` or `MINIMIZE` keywords. However, currently we do not plan to implement this functionality.

It is not possible to define any additional constraints beyond the grammar rules. For instance, the maximum size of solution candidates cannot be constrained; this has to be done in the algorithm configuration instead. A similar constraint would be to limit the maximum number of occurrences of a specific symbol.

We have not yet considered how grammars with optional chains and repetitions can be best transformed to tree-based GP systems. In such systems, the number of children of a node is usually limited and rather small, while grammar rules containing a repetition could potentially lead to a large number of sub-trees.

Another interesting aspect that is not yet supported in GPDL is that a problem definition might also include a-priori knowledge that could be used to guide the search procedure [1]. One idea to specify a-priori knowledge is to add annotations to grammar rules or alternatives to bias search.

Automatically defined functions (ADF) can be included in the grammar for solution candidates as long as the number of ADF is limited and the number of arguments of each ADF are fixed. In this case, it is possible to ensure through grammar rules that functions are only called when they have previously been defined. Architecture altering operators which build function definitions from existing branches and automatically define the number of arguments are not supported, as it would be necessary to defined which symbols can be used to define and call functions in the specified language.

8. ROADMAP

The specification of GPDL and the reference implementation for HeuristicLab are available on our website[3]. On this website we will document relevant discussions on the language specification and changes as well as extensions to the language specification. A next step is to implement a compiler for the popular ECJ framework which is quite similar to HeuristicLab but uses the Java platform. Subsequently, it should also be straight-forward to adapt the compiler to GEVA and possibly JGAP or OpenBEAGLE. Since Coco-2 is available for many languages including C#, Java and C++ it should be possible to translate the compiler definition from the reference implementation to other target languages.

[3] `http://dev.heuristiclab.com/GPDL`

9. REFERENCES

[1] W. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, 68(2):303–366, 1994.

[2] H. Dobler. Top-down parsing in Coco-2. *SIGPLAN Notices*, 26(3):79–87, Jan. 1991.

[3] H. Dobler and K. Pirklbauer. Coco-2: A new compiler compiler. *SIGPLAN Notices*, 25(5):82–90, May 1990.

[4] Y. Hasegawa and H. Iba. Latent variable model for estimation of distribution algorithm based on a probabilistic context-free grammar. *IEEE Transactions on Evolutionary Computation*, 13(4):858–878, Aug. 2009.

[5] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[6] M. Kommenda, G. Kronberger, S. Wagner, S. Winkler, and M. Affenzeller. On the architecture and implementation of tree-based genetic programming in HeuristicLab. In *Proc. of the 14th GECCO*, GECCO Companion '12, pages 101–108, New York, NY, USA, 2012. ACM.

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

[8] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3/4):365–396, Sept. 2010.

[9] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[10] B. J. Ross. Logic-based genetic programming with definite clause translation grammars. *New Generation Computing*, 19(4):313–337, 2001.

[11] 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, 2009.

[12] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.

[13] P. A. Whigham. Grammatically-based genetic programming. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995.