GPML: An XML-based Standard for the Interchange of Genetic Programming Trees

Tiantian Dou, Yuri Kaszubowski Lopes and Peter Rockett
Department of Electronic and Electrical Engineering
University of Sheffield
Portobello Centre
Pitt Street
Sheffield S1 4ET
UK

Technical Report No. CR2018-2r2

26th November 2018

Abstract

We propose a Genetic Programming Markup Language (GPML), an XML-based standard for the interchange of genetic programming trees, and outline the benefits such a format would bring. We present a formal definition of this standard and describe details of an implementation.

1 Introduction

Replicability is a cornerstone of the scientific method that facilitates the independent validation of research findings. Conversely, independent researchers being unable to replicate findings is a means by which erroneous research is rapidly purged from the scientific literature.

A great many of the studies published in evolutionary computing, including genetic programming, are – perforce – both empirical and stochastic, and this imposes a particular difficulty from the point of view of replication. Addressing the genetic programming community in particular, it has become the universal practice to include algorithm parameters such as population size, crossover and mutation rates, etc. in all publications. Although this is welcome, it generally does not give complete information to allow accurate replication of the experiments being reported. For example, complete reporting would require details of the random number generator algorithms employed, their seed values and other details characterizing the stochastic experiments. It could be argued that an adequate number of repetitions of the experiments 'average out' stochastic effects, but the generality of this claim is far from clear.

Adopting standard software for *all* genetic programming (GP) might also be considered to address the replication issue, but we would argue that *independent* implementation is an important aspect of replication. Any bug in the 'standard' software might go undetected for many years, hindering progress in the field. There are cautionary tales here from other disciplines. For example, it has been reported that 1 in 5 of recent genetics research papers contain errors due to formatting problems in the Microsoft Excel spreadsheet program (Washington Post, 2016).

The purpose of this report is to propose a standardised format for the interchange of GP trees. A similar initiative has already been taken on benchmarking problems (White et al., 2013). As a complement to agreed benchmarks, a standard interchange format would offer a number of significant advantages:

1. The publication of the *actual GP trees*, as opposed to a (often partial) description of how the trees were produced, would speed scientific progress in the area.

2. The ability to make direct comparison with the results of other researchers would speed scientific progress, and eliminate the time, frustration and practical difficulties of trying to reproduce others' results, often with incomplete information.

3. Ultimately, the objective is for GP trees to be used as components in larger, complex systems. The coupling of the end-application and the GP training/test software is a major inconvenience, especially in embedded systems. A standard interchange format would allow trained/validated GP trees to be treated as a 'plug-in' component in larger systems.

Since journals, and an increasing number of conferences, provide repositories for supplementary material, it would be straightforward for authors to deposit data fully describing *actual* GP trees that could be accessed by other researchers.

In essence, this paper proposes a standard interchange format based on XML/XML Schemas. This will make it possible for researchers to share the *actual* trees generated during their research thereby exposing them to greater scrutiny/validation by the community as a whole. This should facilitate more rapid progress since researchers will not need to duplicate each other's work in order to make quantitative comparisons with alternative approaches and methods. In addition, GP trees could be treated as a 'plug-in' element for more complex systems, such as robotics; decoupling the training/testing from an application of trained GP trees will have a beneficial effect on the real-world deployment and exploitation of GP.

Fortuitously, a suitable and mature framework for developing a GP interchange format already exists: the eXtensible Markup Language (XML) defined by the World Wide Web Consortium (World Wide Web Consortium, 2008). In Section 2, we briefly describe the relevant features of XML and its accompanying validation framework, XML Schemas. We describe an XML representation of a GP tree in Section 3.

2 Extensible Markup Language (XML)

The eXtensible Markup Language (XML) is a standardised, highly-flexible, human-readable format for information exchange specified by the World Wide Web Consortium (W3C) (World Wide Web Consortium, 2008). Being a text-based representation, XML avoids the hardware-dependent difficulties of binary files. XML itself comprises a series of elements and attributes arranged in a hierarchical fashion – the intrinsic hierarchy in XML lends itself perfectly to describing GP trees, which are, of course, typically represented as acyclic, hierarchical graphs. Further, the syntax of XML is very intuitive although human readability is probably a secondary consideration here since we envisage trees being written/read mainly by computer. Nonetheless, the ability to visually inspect the tree structure is valuable, and often requested by paper reviewers. (We give some simple examples of GP trees represented as XML in Section 3.)

In XML, an element is specified by a syntax such as:

```
<elementName option='A'>
    ...
</elementName>
```

where `elementName` and `option` are user-defined identifiers, and `A` is a quoted text string. Alternatively, an equivalent single-line version where the element does not nest other elements is:

```
<elementName option='A'/>
```

Crucially for the present application, XML elements can also embed other XML elements, as in:

```
<elementName option='A'>
   <otherElementName...>
   </otherElementName>
   <thirdElementName...>
   </thirdElementName>
   ...
</elementName>
```

and so on to arbitrary levels of nesting. XML can thus straightforwardly represent hierarchical structures. See Castro (2001) for a concise but comprehensive introduction to XML.

In a given application, an XML file has to conform to a specified structure. The need for validating the structure of XML files has led to the development of XML Schemas (van der Vlist, 2002; World Wide Web Consortium, 2012) for this purpose. XML Schemas – themselves XML-compliant – are able to specify an XML file structure using a syntax reminiscent of the extended Backus Naur format (EBNF) widely used for specifying the grammar of programming languages; importantly for the present application, XML Schemas are able to specify recursive structures for validation.

In terms of implementation, a large number of proprietary and mature open source XML libraries are available, for example, Xerces (Apache Software Foundation), with bindings to a range of programming languages. In addition, many XML implementations for Matlab are available (e.g. (Mathworks Inc.)). Consequently, there seems no technical impediment to adopting XML as an interchange format.

We term the interchange format proposed here a Genetic Programming Markup Language (GPML) to denote its GP-specialisation over plain XML.

To date, XML has found little application in the GP community. In a rare example, Tanev and Shimohara (2010) have used the Document Object Model (DOM) (WHATWG, 2018), a representation of an XML structure in memory, directly for GP evolution although the DOM is probably not ideal for this purpose.

3 GPML Specification

In this section, we describe GPML, a standardised XML interchange format.

Making use of the intrinsic hierarchy, GPML recursively encodes the tree structure. Different types of nodes and their corresponding information are identified and recorded in various elements and attributes in GPML. By recursively interpreting the element name and attribute information of each node starting from the root node down, the GP tree can be saved and restored.

With elements and attributes arranged in a nested, hierarchical fashion, GPML can be used to describe GP trees intuitively. Typically, the nodes in GP trees can be classified as one of four types: terminal nodes, unary nodes, binary nodes and ternary nodes. These four different nodes correspond to four elements in GPML, namely, `terminalNode`, `constantNode`, `unaryNode`, `binaryNode` and `ternaryNode`, with user-specified information described in the respective attributes.

For the purpose of illustration, consider the simple GP tree shown in Figure 1 that represents the mapping $y = f(\mathbf{x})$ where $y \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^N$ (although there is no restriction with GPML on the input/output being real numbers, or indeed that the elements of $\mathbf{x}$ are even of the same type). This mapping can be straightforwardly represented by the GPML code shown in Listing 1. The intuitively obvious correspondences should be apparent.

In Figure 1, terminal nodes in the GP tree indicate elements of the input vector. We have used the `vectorIndex` attributes of the `terminalNode` elements as indices into the input vector $\mathbf{x}$. For constant nodes, the exact values are specified in the attributes in GPML.

In GP, each unary node has a single child node, which could be any type of node, including another unary node type. In GPML, a unary node is represented by an element `unaryNode`, and its operator is specified by its attribute. Similarly, binary nodes and ternary nodes have two and three child nodes, respectively. In GPML, specific operators are specified by their attributes, like the nodes shown in Figure 1 and their GPML representations in Listing 1.

Note that each GPML document has exactly one root element, which encloses all the other elements. In GPML, the element name of the XML root node is `gpTree`. The single attribute of the root node `noVectorElements` defines the number of regressors used in the mapping, which can be exploited for validation of a GPML document. The indices of terminal nodes in GPML are restricted to the ranges of either $[0 \ldots (\texttt{noVectorElements} - 1)]$ or $[1 \ldots \texttt{noVectorElements}]$, depending on the (implementation-defined) convention adopted for indexing vectors in the actual implementation. Consequently, the attribute `firstIndex` $\in \{‘0’|‘1’\}$ unambiguously associates tree inputs with elements in the input vector. The value of `vectorIndex` in a `terminalNode` element needs to fall in the appropriate range otherwise a validation error occurs that can be straightforwardly detected and notified to the user. It is also a trivial matter to use a tree that has been trained in a system with zero-index vectors in a (separate) system using vectors indexed from unity, and vice versa.

Although the example shown here is for a conventional GP tree, GPML is sufficiently expressive that it could be adapted to other GP variants, such as gene expression programming (Ferreira, 2002), linear GP (Brameier and Banzhaf, 2010), or Cartesian GP (Miller, 2011).

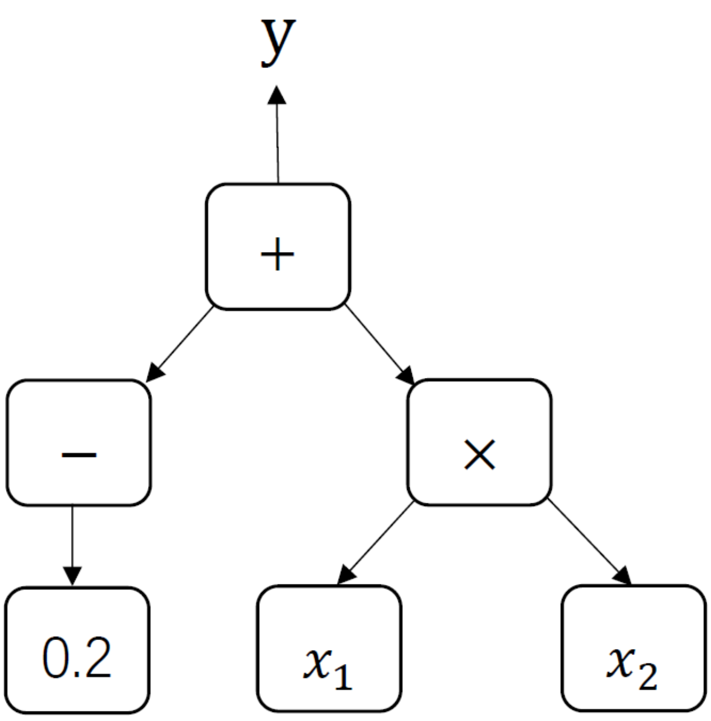

Figure 1: Simple example GP tree.

```
<?xml version='1.0' ?>
<gpTree noVectorElements='2' firstIndex='1'>
    <binaryNode operation='+'>
        <unaryNode operation='−'>
            <constantNode value='0.2'>
        </unaryNode>
        <binaryNode operation='*'>
            <terminalNode vectorIndex='1'/>
            <terminalNode vectorIndex='2'/>
        </binaryNode>
    </binaryNode>
</gpTree>
```

Listing 1: GPML representation of the tree in Figure 1.

In terms of extended Backus-Naur form (ISO/IEC 14977:1996, 1996) typically used to specify languages, we can formally describe the topmost-level `gpTree` element of GPML using:

```
gp-tree = "<gpTree", "noVectorElements=", xml-positive-integer, "firstIndex=", xml-integer ">",
          node,
          "</gpTree>";
```

where the terminal symbol `noVectorElements` is an XML-defined primitive of positive integer type (World Wide Web Consortium, 2012), i.e. a number $\in \mathbb{N}^+$ since the GP mapping is presumed to have at least one input. The non-terminal symbol *xml-positive-integer* value indicates the number of elements in the vector of input variables for the mapping described by the GP tree. As discussed above, `firstIndex`, which takes the values of either 0 or 1, denotes whether the described GP tree indexs the input vector starting from zero or from one.

The non-terminal symbol *node* is defined as by:

```
node = terminal-node | constant-node | unary-node | binary-node | ternary-node;
```

where a *node* is one of either: a terminal node, a constant node, a unary node, a binary node or a ternary node. In their turn, the set of node types, which we believe covers the practical totality of genetic programming, are formally defined by:

```
terminal-node = "<terminalNode", "vectorIndex=", xml-non-negative-integer, "/>";
```

where *xml-non-negative-integer* is again an XML-defined primitive type for an integer quantity ≥ 0 (i.e. $\in \mathbb{N}^0$), and:

```
constant-node = "<constantNode", "value=", xml-double, "/>";
```

where *xml-double* is an XML primitive denoting a double-precision floating-point number.

A unary node is defined by EBNF as:

```
unary-node = "<unaryNode", "operation=", operation,
             [ "parameterString=", parameter-string, ]
             ">",
             node,
             "</unaryNode>";
```

in which *operation* is an XML-defined token type (i.e. a character string), and *parameter-string* is an optional XML token. The `operation` element indicates the (implementation-dependent) operation executed by the unary node; for a unary minus, for example, this might the string '`-`', or for an exponential function '`exp`'. For the cases where a unary node may take some (arbitrary number of additional) parameters, these can be specified using the (optional) '`parameterString`' by concatenating all the function's parameters in, say, a comma-separated list. These parameters can then be simply 'unpacked' by the implementation code. This method of passing any additional parameters is a carefully considered design decision for GPML, striking a balance between simplicity, generality and clarity of GPML syntax. For example, the GP induction of a decision tree (Cao and Rockett, 2015) would typically contain (only) binary nodes that test the state of an element in the input vector and follow the left or right child subtrees, respectively depending on whether the given input vector element was $<$ than or $\geq$ than some threshold. For this application, one possible implementation would be for a node's threshold value to be stored in the optional parameter attribute and extracted by the implementation code.

The *binary-node* and *ternary-node* types in GPML are defined by:

```
binary-node = "<binaryNode", "operation=", operation,
              [ "parameterString=", parameter-string, ]
              ">",
              node,
              node,
              "</binaryNode>";
```

```
ternary-node = "<ternaryNode", "operation=", operation,
               [ "parameterString=", parameter-string, ]
               ">",
               node,
               node,
               node,
               "</ternaryNode>";
```

Yet again, *operation* indicates the operation to be performed by the binary or ternary nodes, respectively, and the optional *parameter-string* conveys any additional parameters required.

The unary, binary or ternary operations specified by the above node types are deliberately left undefined by GPML, and are implementation-dependent. Typically, in a regression problem, the unary operation will be one of: unary minus, exponential function, sine function, etc. Similarly, the binary operations implemented will be one of: addition, subtraction, multiplication, (protected) division, or analytic quotient (Ni et al., 2013), while the ternary operation will typically be `if-then-else`. When learning a boolean problem, on the other hand, the only unary operator would typically be the `NOT` operation, and the binary operations would comprise: `AND`, `OR`, `XOR`, etc. This facility means that GPML can be expanded to include arbitrary, domain-specific operations.

Note also how a `binaryNode`, for example, 'embeds' two *node* types, each of which is defined as being one of a: `terminalNode`, `constantNode`, `unaryNode`, (another) `binaryNode` or `ternaryNode`. The GPML syntax is thus able to recursively define a GP tree of unbounded extent. Similarly, a `gp-tree` 'embeds' a single *node* implying a single root node for the parent tree.

A user is, of course, free to add XML-compliant comments to a GPML file since these will be subsequently ignored by any XML parser.

4 Implementation

Quite deliberately, we do not specify or indeed restrict implementation details for GPML. In this section, we describe our initial implementation as a point of reference.

4.1 Writing GPML

Given a (trained) GP tree in memory, writing a valid GPML file involves a fairly straightforward recursive descent of the tree. With reference to Listing 1, the first task is to output the preamble of the GPML file that comprises the '`gpTree`' information, the '`noVectorElements`' and '`firstIndex`' attributes, and the closing '`>`' character. At this stage, the writing procedure recursively descends the tree (in whatever form this has been implemented) and on 'entering' a node, it emits the appropriate GPML element definition ('`<terminalNode`', '`<constantNode`', '`<unaryNode`', '`<binaryNode`' or '`<ternaryNode`'). Then:

- If the GP node is a terminal (either a '`<terminalNode`' or '`<constantNode`') then it remains only to emit either the '`vectorIndex`' or '`value`' attribute, respectively and a '`/>`' element terminating sequence, and then to return from the recursive call.

- If the current node is a non-terminal GP node, the '`operation`' attribute field together with the optional '`parameterString`' field are emitted, and then the appropriate number of further recursive calls made to visit the child node(s) of the current node. On return from the last of these recursive calls, the function needs to emit a terminating '`</unaryNode>`', '`</binaryNode>`' or '`</ternaryNode>`' field, and then return.

When the chain of recursive calls finally ends, the only remaining task is to emit the '`</gpTree>`' terminating field.

4.2 Reading GPML

We have implemented the initial GPML system using the lightweight `pugixml` XML library[1], largely for simplicity and convenience. The `pugixml` library reads the specified XML file into memory as a Document Object Model (DOM) (WHATWG, 2018), a hierarchical structure that can be traversed

[1] `http://pugixml.org/`

using functions built into the XML library. (Implementation in terms of a DOM is not the only possible approach; the Simple API for XML (SAX) model is equally viable and possibly faster in execution although tends to be more involved in use.) Having created a DOM of the tree, it is a straightforward matter to traverse this data structure, creating and linking GP nodes in an implementation-dependent manner.

4.3 Validating GPML

One of the important elements presented in this work is an XML Schema for the validation of GPML. The simple `pugixml` library we have used does not provide validating facilities although other XML libraries, for example, Xerces (Apache Software Foundation) do although these tend to be more involved to use. In practice, a range of other, convenient validation mechanisms are feasible, for example: the `xmllint`[2] command-line validator, which is part of the `libxml` library . Alternatively, the open-source `jEdit`[3] text editor has an XML plugin that performs validation against a specified schema.

We have made the XML Schema for GPML freely available under a GPL licence on the GitHub repository (`https://github.com/pirlite2/gpml-schema`).

5 Conclusions

In scientific research, reproducibility and transparency are of the utmost importance since science can only progress when the research results produced by an investigator can be independently corroborated by other researchers. The nature of genetic programming research is empirical and stochastic, which imposes particular difficulties on reproducing research findings in the field. We have proposed an XML-based standard for the interchange of genetic programming trees in the hope of speeding scientific progress in the area. GP trees specified in GPML can be evaluated and compared directly by researchers without the effort of having to reproduce others' trees, often with incomplete information.

In terms of implementation, due to its hierarchical structure, GPML can be flexibly represented in XML for which a number of mature, open source XML libraries are available. We have further proposed an XML Schema for the validation of GPML, which is available under a GPL licence on `https://github.com/pirlite2/gpml-schema`.

In addition, in larger systems, a trained-and-validated GP tree can be embedded as a 'plug-in' component using GPML, providing convenience and modularity.

References

Apache Software Foundation. Apache Xerces Project. URL `http://xerces.apache.org/`. [Accessed 25 March 2017].

M. F. Brameier and W. Banzhaf. *Linear Genetic Programming.* Springer, 2010.

Y. Cao and P. I. Rockett. The use of vicinal-risk minimization for training decision trees. *Applied Soft Computing*, 31(0):185–195, 2015. doi: 10.1016/j.asoc.2015.02.043.

E. Castro. *XML for the World Wide Web.* Peachpit Press, Berkeley, CA, 2001.

C. Ferreira. *Gene Expression Programming in Problem Solving*, pages 635–653. Springer London, 2002.

ISO/IEC 14977:1996. Information technology – Syntactic metalanguage – Extended BNF. ISO/IEC. Geneva, Switzerland. 1996.

[2]`http://xmlsoft.org/xmllint.html`
[3]`http://www.jedit.org/`

Mathworks Inc. XML Documents. URL `http://www.mathworks.com/help/matlab/xml-documents.html`. [Accessed 25 March 2017].

J. F. Miller, editor. *Cartesian Genetic Programming.* Springer, 2011.

J. Ni, R. H. Drieberg, and P. I. Rockett. The use of an analytic quotient operator in genetic programming. *IEEE Transactions on Evolutionary Computation*, 17(1):146–152, February 2013. doi: 10.1109/TEVC.2012.2195319.

I. Tanev and K. Shimohara. XML-based genetic programming framework: Design philosophy, implementation, and applications. *Artificial Life and Robotics*, 15(4):376–380, 2010.

E. van der Vlist. *XML Schema.* O'Reilly, Sebastopol, CA, 2002.

Washington Post. An alarming number of scientific papers contain Excel errors, 2016. URL `https://www.washingtonpost.com/news/wonk/wp/2016/08/26/an-alarming-number-of-scientific-papers-contain-excel-errors/`. [Accessed 27 March 2017].

WHATWG. Document Object Model (DOM), 2018. URL `https://dom.spec.whatwg.org/`. [Accessed 25 June 2018].

D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke. Better GP benchmarks: Community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.

World Wide Web Consortium. Extensible markup language (XML) 1.0, 2008. URL `https://www.w3.org/TR/2008/REC-xml-20081126/`. [Accessed 17 July 2018].

World Wide Web Consortium. XML Schemas, 2012. URL `https://www.w3.org/standards/xml/schema`. [Accessed 17 July 2018].