XML-based Genetic Rules for Scene Boundary Detection in a parallel processing environment

Minaz J. Parmar Prof. Marios C. Angelides
Brunel University, Uxbridge - London, UK
{minaz.parmar, marios.angelides}@brunel.ac.uk

Abstract

Genetic programming is based on Darwinian evolutionary theory that suggests that the best solution for a problem can be evolved by methods of natural selection of the fittest organisms in a population. These principles are translated into genetic programming by populating the solution space with an initial number of computer programs that can possibly solve the problem and then evolving the programs by means of mutation, reproduction and crossover until a candidate solution can be found that is close to or is the optimal solution for the problem. The computer programs are not fully formed source code but rather a derivative that is represented as a parse tree. The initial solutions are randomly generated and set to a certain population size that the system can compute efficiently.

Research has shown that better solutions can be obtained if 1) the population size is increased and 2) if multiple runs are performed of each experiment. If multiple runs are initiated on many machines the probability of finding an optimal solution are increased exponentially and computed more efficiently. With the proliferation of the web and high speed bandwidth connections genetic programming can take advantage of grid computing to both increase population size and increasing the number of runs by utilising machines connected to the web. Using XML-Schema as a global referencing mechanism for defining the parameters and syntax of the evolvable computer programs all machines can synchronise ad-hoc to the ever changing environment of the solution space.

Another advantage of using XML is that rules are constructed that can be transformed by XSLT or DOM tree viewers so they can be understood by the GP programmer. This allows the programmer to experiment by manipulating rules to increase the fitness of a rule and evaluate the selection of parameters used to define a solution.

Introduction

Evolutionary computing has spawned numerous types of methods for automatically generating solutions to problems that are to complex and intricate to solve by conventional methods. It is based on processes of natural evolution proposed by Darwin where he states that natural biological organisms must evolve in order to gain an advantage in an environment where other organisms exist to be able to survive and propagate [1]. This genetic “arms race” leads to an ethos of survival of the fittest. Genetic programming (GP) is an automatic programming method that finds the best fit solutions (or programs) by processes that are analogous to those proposed by Darwin for evolution [2].

Genetic programming has been used to solve many complex problems where the amount of solutions possible is almost infinite and to compute all possible

solutions is impractical or impossible. The main difference between GP and other types of evolutionary computing is the use of parse trees. GP is implemented by defining sets of functions and terminals that are structured using a formal syntax that will represent the programs to be evolved. These functions and terminals can be represented in a tree like structure that consists of nodes and leafs. The nodes are the functions and the leaves the terminals. This method strips away the rather unnecessary grammar of raw source code and leaves only the essential operating data. The reason for this is if raw source code was used to represent a computer program to be evolved the interactions required for mutation, reproduction and crossover would be complex and extremely error prone. This representational flexibility allows genetic programming to solve a wide range of problems.

In this particular case GP was used to automatically detect scene boundaries from AV source [3]. Detecting scene boundaries is difficult as it is a matter of human perception where a scene ends and finishes. This requires a certain degree of semantic awareness of the content rather then just deciphering low-level syntactical features to provide audio and visual cues such as in shot boundary detection [4, 5]. Most scene boundary techniques are genre based as they use combinations of visual and audio cues that are uniform and homologous to the genre. Unfortunately rules devised in such a manner only hold true for that genre and cannot be easily applied to other genres with the same degree of accuracy. The GP scene boundary detection algorithm was generic in its approach to scene boundary detection because it would devise a rule that was tailor made for a particular clip of AV content that would give a high degree of precision and recall of the scene boundaries. Using this approach to scene boundary detection allows the process to be applied generically to any content without concern for the genre of the material.

The probability of finding an optimal solution from a population increases with an increase in size of the population and the amount of runs performed to evolve an optimal solution. The drawbacks to increasing the population are computational efficiency of the algorithm, i.e. the larger the population size the longer it will take to evaluate the solutions to find the optimal solution.

In this paper we discuss the creation of genetic programming rules for scene boundary detection that can be processed in parallel by any number of systems that use grid computing to increase the computational effectiveness and efficiency genetic programming. Using the XML-Schema specification [6] we define the functions, terminals and syntax used for the rules and then bind the functions and terminals to java data types using JAXB. They are then evolved to produce scene boundary detection rules. The rules are also human readable so they can be analysed and modified manually to see what effects it has on the fitness of a rule.

Parallel processing

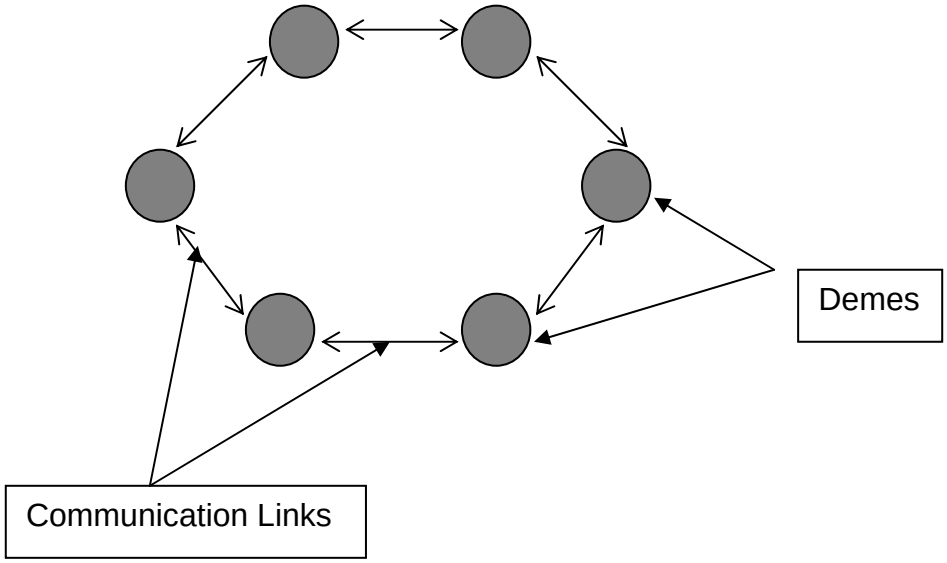
A solution to the population vs. computational efficiency problem is to employ a parallel processing model (sometimes called the farmer or master-slave model) that can be used so that multiple populations can be initiated and then evolved in synchronisation. The drawback is the overhead in communication grows with the amount of populations produced [7]. By dividing the population into more independent subpopulations, two alternative parallel models can be identified [8].

Based on the size and number of subpopulations, they are referred to as coarse-grained or fine-grained distributed population models. When dealing with very large populations, which are common in hard, human-competitive problems, these models are better suited since their overall communication capacity scale better with growing population size.

One popular method is the Island model (see figure 1). The coarse-grained, distributed population model consists of a number of subpopulations or “demes” that evolve rather independently of each other. With some migration frequency they exchange individuals between each other over a communication topology.

The island model is a very popular parallel model, mainly because it is very easy to implement on a local network with standard workstations (cluster) [9] but can be adapted to work on a larger networks such as the web. The communication between demes is infrequent and therefore a global control must be found to regulate changes and communication between demes.

Figure 1. Island Model



Using XML-schema to define the parameters of the GP algorithm allows all demes to be controlled by addressing a global referencing mechanism which allows them to modify their local parameters according to the global schema. This ensures uniformity of in the construction of the initial populations, regulates evolving populations and makes them robust to parameter changes affecting the algorithm.

Using a parallel architecture also ensure that there is less chance of a premature convergence of the population which leads to a better quality solution for the problem being produced.

Human readable solutions

The original scene boundary algorithm used rules that were written in RPN (reverse polish notation). These rules can be very complex and hard to read. In natural genetics there are a lot of redundant genes that play no active part in cell growth. The same can be said of GP where redundant data in a rule can be omitted making the rule

more efficient at solving the problem without reducing the accuracy. GP's main drawback is the computational efficiencies of the programs evolved. The principle of parsimony supports this approach and is also at the root of a GP programmer's efforts. Also a rule can be improved if its structure is modified (i.e. a function(s) or terminal(s) is added or removed) to give it a better fitness value. With rules written in RPN editing a rule was very difficult.

One of the advantages of genetic programming is that you can analyse any solution in any given generation as it is a compact form of the functions and terminals of a program. Usually though these cannot be read by a humans as the rules become long and complex. Most rules are written in the manner of RPN. Identifying sub-computations within it are impossible without first interpreting the whole solution first.

The ability to understand the solutions presented is helpful in engineering new parameters that might be required to solve the problem more effectively. The evolution of programs might be automatic but the choice of parameters is not. This is effectively what a GP programmer main duty is, to ameliorate the set of parameters to be encoded in order to solve the problem more effectively and efficiently.

Representing rules in XML makes them readable by parsing the structure into a DOM tree. This makes human interpretation of the rule easier. Nodes can be collapsed and expanded to show the sub computations and the relationship between them and the complete structure. Values can be altered more easily as their relevance is more clearly understood. Human readable solutions will also lead to a better understanding of the parameters they have chosen to represent the computer program they wish to evolve. This multi-view abstraction of the solutions allows programmers to decide on what parameters are required to best solve the problem.

Implementation

We believe that defining the parameters of the rules using XML-schema is an implicit and robust method suited to encoding solutions for GP algorithms. Below we discuss how it was implemented for using in the generation of scene boundary rules as the example scenario. This method of encoding rules is easily transportable to all uses of genetic programming when encoding parameters for GP parse trees.

XML-Schema and GP

The difference between GP and other forms of evolutionary computing is program representation. GP represents its computer programs as population of trees. The nodes of the trees represent functions whilst the leaves represent the terminals that hold the data to be computed. Trees have arbitrary sizes and structures [10]. There are three popular types of tree structure; 1) *full* 2) *grow* and 3) *ramped half-and-half* methods. The *full* generative method creates a population with full trees. The *grow* generative method on the other hand, generates the initial population with trees that are variably shaped. The *ramped half-and-half* method is a hybrid of both the full and grow methods.

The standard single-typed genetic programming system operates using an abstraction of computer programs (an already parsed expression), typically

represented in a parse tree. The use of a parse tree representation in a genetic algorithm was pioneered by Cramer [11]. Parse trees serve no other function other than that it circumvents issues of a purely syntactical nature and suggest a few natural variation operators. The number of arguments for each function can be deduced from the number of children of a node and also issues of operator precedence are resolved. The parse tree thus represents an unambiguous way of computing a solution. It is this property that is also employed by compilers. These generally use parse trees as an intermediate representation before generating machine code. The parse tree also provides inspiration to the issue of variation. As a parse tree decomposes a computation into a hierarchy of sub-computations, varying these sub-computations at the various levels in the tree is a natural way of obtaining new programs.

XML [12] is perfect at representing parse trees because both their syntactical structures are analogous. Elements in XML can be the nodes and leafs; therefore they can represent the functions and terminals represented in a parse tree. An XML element can nest another element or elements inside of it, supporting a parse trees function of supporting hierarchies of sub-computation. The sub-computation represented by XML can be varied by simply exchanging or altering an element. XML tree can have any arbitrary structure and can therefore support all methods of tree structure generation; full, grow or ramped half-and-half.

The only issue arises to how the structure of an XML document can simulate the syntax of a parse tree. A rule (solution to the problem of scene boundary detection) is represented by a syntactically legal symbol sequence with every symbol being an element of either a function set (F) or a terminal set (T) that both underlie a genetic-programming approach. If, for instance, the syntax of arithmetic expressions is given, then a + b and a are legal symbol sequences constructed from the sets F = {+} and T = {a, b}. Thus, the solution space is the set of all legal symbol sequences.

XML was created as a “does anything” language where its elements can represent any arbitrary value. This particular point gains more emphasis when using parallel processing of GP as the parameters are subject to change and the changes should be compatible with the other demes. In order that all computations are structured in formal legal symbol syntax a grammar for the rules must be defined. Firstly the function set must be defined and then the terminal set. These are all defined as elements within the document. The relationship between the elements is defined to provide the syntax of the solutions. In XML schema the type definitions of the elements can be specified when the element is declared.

Specification of parameters

Defining the parameters to a solution is an important task in the success of an GP algorithm. The most important lesson is that one must understand the problem in order to be able to properly use the GA algorithm [13]. To this end we have selected the following functions and terminals to find a solution to the scene boundary detection problem. The feature set to be used to be used as the main parameters of the GP algorithm are:

- Shot Duration – the length of a shot in seconds till the start of the next shot

- Histogram difference – The change in the mean histogram values of the RGB values of the first frame of a shot to a specified preceding/subsequent shot

- Transition Effect – What transition effect there is between the shot (gradual or cut transitions?)

- Audio break – is there an audio break between the shots? If there is what type of audio was preceding it (music or speech?)

Defining the functions and terminals

The function set of the algorithm can be defined as $F = \{SD, HD, TE, AB, AND, OR\}$, Where *SD*, *HD*, *TE*, *AB* are the four features; *Shot duration*, *Histogram difference*, *Transition effect*, *Audio Break* respectively and AND, OR are Boolean operators. The terminal sets comprise of $T = \{sp, bv, pi, op1, op2\}$, where sp = {A, B, C, D, E} is the position of the shot to be compared against the current shot (C), *bv* = {true, false}, *pi* is a positive integer in the range of 1 – 126789, $ops1 = \{=, \neq\}$ for Boolean operations and $ops2 = \{<, \geq\}$ for arithmetic operations. The formal symbol syntax is shown in table 1.

Table 1: Formal symbol syntax

The tree root must be either AND or OR
The left child of a *TE*, *HD*, *SD* or *AB* must be a *sp*
The middle child of a *TE* or *AB* must be an *ops1*
The middle child of a *HD* or *SD* must be an *ops2*
The right child of a *TE* or *AB* must be a *bv*
The right child of a *HD* or *SD* must be a *pi*

Encoding the parameters into XML-Schema

XML-schema is ideal for encoding the parameters and their syntax because:

- defines elements that can appear in a document
- defines attributes that can appear in a document
- defines which elements are child elements
- defines the order of child elements
- defines the number of child elements
- defines data types for elements and attributes
- defines default and fixed values for elements and attributes

Each generation of a population (initial and evolved) is created into an XML document. The advantage of this is that each individual generation can be stored and then accessed by other processing stations. Each generation is marked with the ID of the station that generated it along with the generation of the population. The last attribute is the size of the population of that generation. Embedded in the generation node is the rule element, which will represent one completely formed rule. The maximum occurrence of the rule element is not specified as this will be governed by the population attribute.

Figure 2. The XML Schema definition of the scene boundary detection rules

```
    <xsd:annotation>
    <xsd:documentation xml:lang="en">
    Scene boundary detection parsing rules schema for genetic programming Algorithm.
    Copyright 2004 Minaz Parmar. All rights reserved.
    </xsd:documentation>
    </xsd:annotation>

    <xsd:element name="Generation" type="GenerationType"/>

    <xsd:complexType name="GenerationType">
        <xsd:sequence>
            <xsd:element name="Rule" type="RuleType" minOccurs="1"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="ID" type="xsd:positiveInteger"/>
    </xsd:complexType>

    <xsd:complexType name="RuleType">
        <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element name="AND" type="BooleanOperator" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="OR" type="BooleanOperator" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="IF" type="BooleanOperator" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="NOT" type="BooleanOperator" minOccurs="1" maxOccurs="1"/>
        </xsd:choice>
    <xsd:attribute name="ID" type="xsd:positiveInteger"/>
    </xsd:complexType>

    <xsd:complexType name="BooleanOperator">
        <xsd:all minOccurs="0" maxOccurs="1">
            <xsd:element name="AND" type="BooleanOperator" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="OR" type="BooleanOperator" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="IF" type="BooleanOperator" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="NOT" type="BooleanOperator" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="TransitionEffect" type="TransitionFeatureType"
minOccurs="0" maxOccurs="1"/>
            <xsd:element name="AudioBreak" type="AudioBreakFeatureType" minOccurs="0"
maxOccurs="1"/>
            <xsd:element name="HistogramDifference"
type="HistogramDifferenceFeatureType" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="ShotDuration" type="ShotDurationFeatureType"
minOccurs="0" maxOccurs="1"/>
        </xsd:all>
    </xsd:complexType>

    <xsd:complexType name="TransitionFeatureType">
        <xsd:sequence>
            <xsd:element name="ShotPosition" type="ShotPositionType" minOccurs="1"
maxOccurs="1"/>
            <xsd:element name="Operator" type="BooleanOpType" minOccurs="1"
maxOccurs="1"/>
            <xsd:element name="Value" type="xsd:boolean" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="type" type="TransitionType"/>
    </xsd:complexType>

    <xsd:complexType name="AudioBreakFeatureType">
        <xsd:sequence>
            <xsd:element name="ShotPosition" type="ShotPositionType" minOccurs="1"
maxOccurs="1"/>
            <xsd:element name="Operator" type="BooleanOpType" minOccurs="1"
maxOccurs="1"/>
            <xsd:element name="Value" type="xsd:boolean" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="type" type="AudioBreakType"/>
    </xsd:complexType>

    <xsd:complexType name="HistogramDifferenceFeatureType">
        <xsd:sequence>
            <xsd:element name="ShotPosition" type="ShotPositionType" minOccurs="1"
maxOccurs="1"/>
            <xsd:element name="Operator" type="IntegerOpType" minOccurs="1"
maxOccurs="1"/>
            <xsd:element name="Difference" type="xsd:positiveInteger" minOccurs="1"
maxOccurs="1"/>
        </xsd:sequence>
```

```
    </xsd:complexType>
    <xsd:simpleType name="BooleanOpType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="!="/>
                    <xsd:enumeration value="="/>
                </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="IntegerOpType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="&gt;"/>
                    <xsd:enumeration value="&lt;"/>
                    <xsd:enumeration value="=&gt;"/>
                    <xsd:enumeration value="&lt;="/>
                    <xsd:enumeration value="=="/>
                </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

The AND /OR elements are the root elements for a rule and are restricted to appear once exclusively at the top of the rule as stated in the syntax specification. The AND/OR elements are of type “Boolean Operator”. In this defined type we have the AND, OR, TE, AB, HD and SD elements as possible child nodes that can appear only once or not at all. This structuring of the AND/OR elements supports recursion that is required for making trees of different depths. The definition of the AND/OR element ensures that rules can be of any arbitrary size and structure supporting all generative methods. The TE, AB, HD and SD are terminating nodes that contain leaf elements. Each feature falls into one of two categories that define what operations can be performed on them; Boolean or arithmetic. The TE and AB have child elements that correlate to the terminal sets sp, ops1 and bv. The HD and SD have child elements that correlate to the terminal sets sp, ops2 and pi.

Figure 2 presents an extract from the schema showing the more important element definitions. It should be noted that the definition of the function nodes are user defined types that contain complex type definitions whilst the terminal set represented by the leaves are primitive types (Boolean, positive integer and string). This correlates to parse tree structure where the function nodes serve the purpose of relationships and operations to be performed on the terminal set.

The feature types are defined so that the elements will appear in a strict sequence that cannot be deviated from. This ensures that the data will be readable to the human eye as the grammar will always be uniform. Attributes are used mainly for as unique identifiers but are sometimes used to give an element a property that child elements will refer to e.g. the TE element has attribute “type” that relates to either gradual or cut transitions. These attributes are enumerated so they cannot be deviated from to guarantee consistency of the syntax throughout the demes.

The child elements of all the feature types are simple type definitions, that is they contain no child elements and are derived from simple data types; string, positive integer and Boolean. These can be mapped directly to java data types in the binding process.

Binding the XML-Schema based rules using JAXB

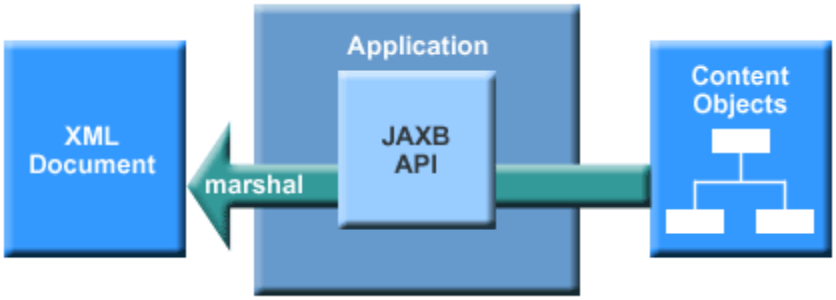
The JAXB API (Java Architecture for XML Binding) [14] provides a convenient way to bind an XML schema to a representation in Java code. This makes it easy for you to incorporate XML data and processing functions in applications based on Java technology without having to know much about XML itself. The JAXB binding compiler takes XML schemas as input, and then generates a package of Java classes and interfaces that reflect the rules defined in the source schema. These generated classes and interfaces are in turn compiled and combined with a set of common JAXB utility packages to provide a *JAXB binding framework*. This is commonly referred to as the Java content tree.

In our scenarios the complex types such as the Rule and Feature types that represented the functions become classes that hold variables of primitive type that represent the terminals. The terminal data types are matched from XML-Schema defined types to java data type representations e.g. the positive integer type of XML-Schema is translated into the Big integer class which is a standard class for holding large positive numbers.

Figure 3. Binding the schema [15]

We use this framework by generating a Java content tree that represents a generation of a population of rules. The Java content tree is then marshalled into a XML document that can then be unmarshalled by another application using the same Java content tree. Any modifications to the XML schema can be ratified without having to marshal the tree into XML, which supports validation on demand.

Figure 4. Marshalling the content tree [15]

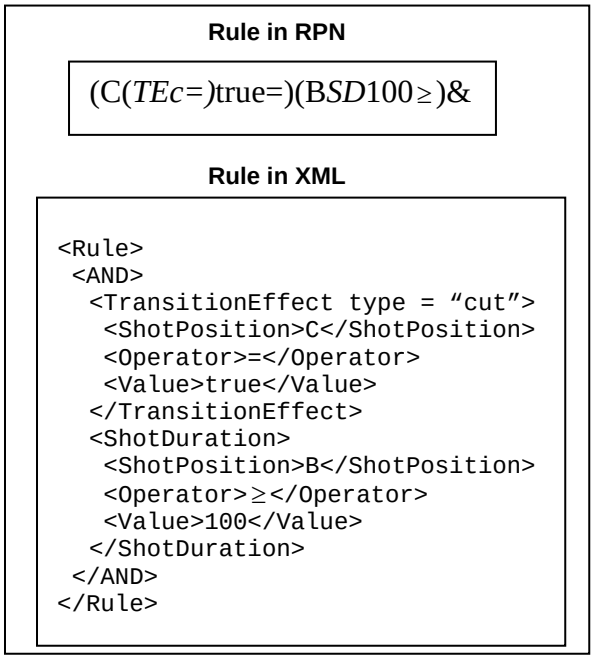


Advantages of XML based rules

The main advantage to the rules was the human readable aspect of the XML derived rules. Shown in Figure 3 is a rule represented in RPN and XML. The example is simple but shows the problems with RPN, It is hard to visualise the relationship between the function and the terminals with the way it is written. When this experiment was originally carried out the authors found that the best fit rule would

miss a scene boundary by a shot. What they wanted to do was tweak the rule to see if they could achieve a greater precision. By this type of experimentation they could identify requirements for modification or addition of parameters for functions and terminals. With XML based rules a DOM tree viewer can be used to analyse clearly the relationship and structure of a rule. Alterations can be made with clear insight into what affect they will have on the solution.

Figure 3. A rule in RPN and XML



When using parallel processing functions and terminals can be added by changing the definitions in the XML-schema. All machines running the experiment can follow changes to the GP parameters by using the schema as a global reference to identify what functions and terminals are being used and what the syntax is to construct a well formed solution. This is suitable for the island model where communication is infrequent between demes. If a parameter change is required the schema definition can be updated therefore all demes will be notified of the alterations.

Using XML-schema to represent parse trees is an intuitive implementation strategy for GP as the syntax of the rules is implicitly stated in the definition of the elements that represent the functions and nodes. Using JAXB to parse and validate the schema into java data types ensures that all rules generated will be well formed without any need to explicitly check for correct syntax.

Conclusion

We have examined the advantages of using XML-schema to derive XML based parse trees that support the use of parallel-processing GP models. They can be and aide to programmers in the discovery and manipulation of parameters, illuminating new strategies by helping the programmer visualise the relationships between structure and function of possible solutions. Although this has been shown in

the context of generating scene boundary detection algorithms it can be applied to all examples of GP applications.

Using XML-schema to define the rules could allow for global collaboration of GP experiments as XML-schema allows not only global access but translation from one language to another and the use of multiple schemas to allow experiments to have different variations on parameters.

In our research there is a small increase in the time taken to initial parse the rules from XML into java data types then was taken by RPN, but once that is done the computational time is almost identical compared to RPN. This small initial overhead is well worth the trade off for the benefits XML based rules provide.

Further work must be carried out on using XML schema to communicate changes in the basic genetic algorithm when it introduces new parameters, for instance the migration policy and the network topology. Today, there exists little or no theory on how to adjust those parameters.

References

[1] C. Darwin, “On the Origin of the Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life”. 1859.

[2] J. R. Koza, “Genetic Programming: On the Programming of Computers by Natural Selection”, Cambridge, MA, USA, MIT Press, 1992.

[3] T.S.K. Lo and M.C. Angelides, “A video content independent mining algorithm for evolved rule-based detection of scene boundaries”, Ingenierie des Systemes d'Information Journal, forthcoming, 2004.

[4] Z. Rasheed, M. Shah,”Scene Detection In Hollywood Movies and TV Shows”, In Proceedings of 2003 Conference on Computer Vision and Pattern Recognition (CVPR '03), Volume II, pp. 343, June 2003.

[5] D. Zhong, S. Chang, “Structure Analysis of Sports Video Using Domain Models”, In Proceedings of 2001 IEEE International Conference on Multimedia and Expo, pp. 182, August 2001

[6] XML-Schema Part 0: Primer,
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/,
Last accessed 18/10/2004.

[7] Bethke, A. D, “Comparison of Genetic Algorithms and Gradient-Based Optimizers on Parallel Processors : Efficiency of Use of Processing Capacity”, Tech. Rep. No. 197, University of Michigan, Logic of Computers Group, Ann Arbor, MI, 1976.

[8] S. E. Eklund, Time Series Forecasting Using Massively Parallel Genetic Programming, In Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'03), pp. 143a, April 2003

[9] E. Cantú-Paz, “A Survey of Parallel Genetic Algorithms”, Department of Computer Science, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 1998.

[10] S. Luke, L. Panait, A survey and comparison of tree generation algorithms, In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pages 81-88, San Francisco, California, USA. Morgan Kaufmann, 2001.

[11] N. L. Cramer, “A representation for the adaptive generation of simple sequential programs”. In Proceedings of an International Conference on Genetic Algorithms and the Applications, pages 183-187, Carnegie-Mellon University, Pittsburgh, PA, USA, 1985.

[12] Extensible Markup Language (XML) 1.0 (Third Edition), http://www.w3.org/TR/2004/REC-xml-20040204/,
Last accessed 18/10/2004.

[13] N. Melvin, R. Soricone, J. Waslo, “On the Automaticity of Genetic Programming”, In Proceedings of the 14th International Conference on Electronics, Communications, and Computers (CONIELECOMP 2004), pages 236-242, IEEE Computer Society, ISBN: 0-7695-2074-X, Veracruz, Veracruz, México. February, 2004.

[14] JAXB homepage, http://java.sun.com/xml/jaxb/, Last accessed 26/10/04.

[15] Article on Java Architecture for XML Binding (JAXB),
http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html,
Last accessed 26/10/04.