IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 3, 2010
ISSN (Online): 1694-0784
ISSN (Print): 1694-0814

Domain-Independent Genotype to Phenotype Mapping through XML Rules

Daniela Xhemali[1], Christopher J. Hinde[2] and Roger G. Stone[3]

[1] Computer Science, Loughborough University,
Loughborough, LE11 3TU, UK
D.Xhemali@lboro.ac.uk

[2] Computer Science, Loughborough University,
Loughborough, LE11 3TU, UK
C.J.Hinde@lboro.ac.uk

[3] Computer Science, Loughborough University,
Loughborough, LE11 3TU, UK
R.G.Stone@lboro.ac.uk

Abstract

This paper discusses an innovative approach to mapping Genotypes to Phenotypes through XML rules. Specifically, it concentrates on the mapping process using two very different domains – Regular Expressions (REs) and Software Program Statements. The paper shows that our Genotype-Phenotype system can be applied to any domain that requires the use of REs and it can be adapted to work for any other domain with minimum effort.

Keywords: *Genetic Evolution, Genotypes, Phenotypes, XML Mapping, Regular Expressions, Complete Software Structures.*

1. Introduction

GP research has attracted attention in various fields such as: game strategies [14], military defence [13], plant biology [7], electronics [20], railway platform allocation [4], spam filtering [5], feature extraction from media files [12], [16], automated web extraction [3], [31] etc. One can use different terminologies to define GP, but fundamentally: *"At the most abstract level, GP is a systematic, domain independent method for getting computers to automatically solve problems starting from a high-level statement of what needs to be done"* [18].

This paper focuses on a specific part of GP – the Genotype-Phenotype Mapping – thus other components of GP will not be covered here. Readers are encouraged to look at [31] for a discussion of the different GP components in relation to a real project, as well as [1], [8] and [9] for a large selection of GP papers and surveys of the history of evolutionary computing.

The Genotype-Phenotype Mapping relates to the way individuals in a population are represented, as this can have a significant effect on the performance of GP. A Genotype represents each individual in the search space, whereas its Phenotype represents the individual in the solution space [2]. Some research, particularly earlier GP research, do not make a distinction between Genotypes and Phenotypes [5], [17], [27]. Individuals in each genetic population remain the same throughout the evolution process. In these works the search space and the solution space are identical.

In 1994, Banzhaf [2] suggested the separation of the two spaces and introduced his work on the Genotype-Phenotype mapping. The separation involves the encoding of the individuals to a form known as the 'Genotype', which is later on decoded back to the corresponding program, referred to as the 'Phenotype'. This separation simplifies and increases the efficiency of certain genetic operations such as: reproduction and mutation, because these would no longer be constrained by the parameters used in the program being evolved. In Genotype-Phenotype based GP, genetic operators such as Crossover and Mutation would be performed on the Genotype, whereas other processes, such as the Fitness scoring, would be performed on the Phenotype. Sections 3.1 and 3.4 explain this concept further through examples.

There are researchers who criticise the separation into Genotypes and Phenotypes [19]. The main concern expressed is that the conversion process of a mutated

Genotype into the Phenotype may result in anomalies that could potentially lead to invalid solutions. A direct mapping between the encoded program and the actual program is therefore vital to ensure the validity of the solutions [23], [28].

In our research, the Genotypes are presented as strings of integers. The direct mapping of these integers to the corresponding structures is achieved through an innovative approach involving XML rules as described in section 3.4.

This research is part of a larger project in collaboration with an independent brokerage – Apricot Training Management (ATM) – which helps organisations to identify and analyse their training needs and recommend suitable courses for their employees. ATM currently uses time-consuming, labour-intensive, manual techniques to gather information related to training courses, however, this process often results in out-of-date courses being stored in the company's database and many hours being wasted on Web browsing and data entry. Our overall project is to provide ATM with a system that will automate the extraction and storing of training course information into the company's database, guaranteeing always up-to-date training data [29], [30]. Specifically, the research concentrates on the evolution of Regular Expressions (REs) for the extraction of pieces of information such as course names, prices, dates and locations from training web pages.

The following section focuses on research and techniques related to Genotype-Phenotype based GP.

2. Related Research

Many researchers have embraced the separation of the search space from the solution space through the use of Genotypes and Phenotypes [2], [15], [28]. This, however, adds an additional step to the genetic evolution process – the translation or mapping of the Genotypes to their corresponding Phenotypes. This step occurs after the genetic reproduction stage (i.e. the crossover and mutation) and before the Fitness test can take place.

The following concentrates on the different methods that have been used to achieve the mapping process.

Banzhaf [2] represented his Genotypes as linear binary strings. The mapping stage then processed these Genotypes from left to right in 5-bit sections, where each 5-bit code mapped to a pre-specified symbol. For example: 00000 mapped to PLUS, 00100 mapped to POW, 11000 mapped to variable X etc. The first bit indicated whether the code represented a function (PLUS, POW etc.) or a terminal (X, Y etc.). The research also discussed their concern about generating constant numbers. Koza [17] solved this problem by defining "random ephemeral constants" where constants are only generated once for a particular program and then reused wherever they are needed within that program.

Keller [15] continued in the footsteps of Banzhaf, concentrating on providing experimental evidence for choosing the Genotype-Phenotype approach instead of the normal GP approach. Keller's system however, could only evolve programs in languages defined by the LALR (Look Ahead Look Recursive) grammar, as this was the grammar chosen for the repairing stage of the Genotype-Phenotype mapping process.

There was a certain amount of redundancy in the genetic coding in both Banzhaf's and Keller's works. They both admitted that, in their works, different binary strings could correspond to the same symbol, which could lead to inconsistencies e.g. 000 and 100 both mapping to 'a'.

A slightly different Genotype representation is seen in the work of Withall et al. [28]. In here Genotypes are represented as linear blocks of integers. Each block comprises exactly four integers, each representing a different gene. Although both research works used fixed-length genomes, in [2] the resulting Phenotypes could vary in length, whereas in [28] they remained fixed.

The first integer in Withall's Genotype determines the type of function that follows. Similarly, the Genotype in our research is represented as a string of integers. There are no fixed length genomes determined however; instead the Genotype can contain any number of genes.

The unique feature of our research is the use of XML to define the necessary rules to achieve the Genotype-Phenotype mapping. The first gene in the string determines the XML rule to be followed, which in itself guides the mapping of the rest of the genes into a valid Phenotype. This is explained in detail in section 3.4.

3. Genotype to Phenotype Mapping

Before discussing our mapping approach, it is important to explain a few related topics in order for the reader to fully understand how the approach works. The following discusses the main representation chosen for the system, as well as the different domains on which this approach was tested. Specifically, the two domains relate to the Genotype-Phenotype mapping of: REs and Software Program Statements. The Software Program Statements domain was chosen because it is very different from that of REs and thus, it can truly illustrate the extent to which our system needs to be changed to apply to such a domain.

Section 3.4 explores the XML mapping process in detail.

3.1 GP Representation

There are two main representations in GP: the Tree-Based GP and the Linear GP [28]. The Tree-Based GP represents programs as syntax trees (Fig. 1), where program variables and constants make up the tree leaves (x, 1, 5, y, 2), whereas the program operators (*, +, –, /) are the internal nodes. The tree leaves are also known as the terminals, whereas the internal nodes are known as the functions. Fig. 1 shows the tree representation of program "(x+1)*(5-(y/2))". For complex programs, the main tree may contain many sub trees.

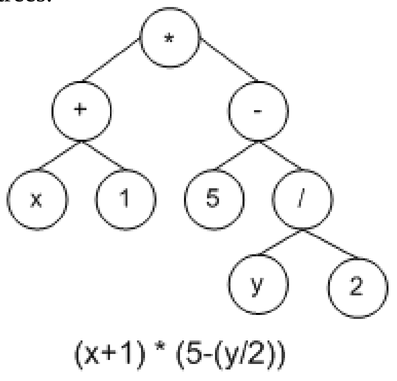


Fig. 1 Tree-based GP Representation

Linear GP represents programs as a linear sequence of instructions (Fig. 2). Based on biological evolution, the sets of instructions are known as Genomes and each individual instruction is called a gene. All the available Genomes for a particular program form the Genotype. Linear GP representations may have either fixed length Genomes i.e. the same number of genes for every instruction, or variable ones. This depends on the problem to be solved.

1st Instruction	2nd Instruction	...	nth Instruction

Fig. 2 Linear GP Representation

We have chosen the linear approach for our research, because it is more appropriate for the representation of non-standard models like REs. Furthermore, linear GP runs faster than tree-based GP [21]. This is because almost all computer architectures represent computer programs in a linear fashion – as a sequence of commands to execute. The execution of tree-shaped programs is not natural for computers, thus interpreters or compilers would need to be used as part of the tree-based GP [21], which is computationally expensive.

3.2 Regular Expressions (REs)

REs [26], [10], [22], [25] are powerful tools used to detect patterns in data. They can range from basic to very complex, matching from just literal text[1] to very specific instances of text based on certain criteria. For example: ^*[A-Z][a-z]+* matches all instances that begin (^) with an uppercase letter (*[A-Z]*) followed by one or more (+) lowercase letters (*[a-z]*) such as "Regular" or "Expression" but not "RE" or "REs".

REs are very well known, particularly in the UNIX community and they feature largely in some programming languages such as Perl, PHP or AWK. However, the manual generation of REs can be a difficult, error-prone and time consuming undertaking, especially for complex patterns. This is due to the fact that although REs are built up from small building blocks, where each block is fairly simple; all the available blocks can be combined in an infinite number of ways [10], which may result in a highly complex RE. Tools have been developed to evaluate the validity of REs [11], [24], however, very little, if anything has been done towards the automatic generation of REs.

Our research focuses on the automatic generation of REs through the use of genetic programming principles in order to automate web extraction. This paper concentrates on the mapping of Genotypes (strings of meaningless numbers) to Phenotypes (REs) through XML rules as explained in section 3.4.

3.3 Software Program Statements

The research in this paper is based on the work of Withall et al. [28]. The examples are kept as close to the original as possible in order to ensure their integrity. The only difference is that the Software Program Statements in [28] were evolved to be valid in Perl, whereas in this paper their validity is ensured against VB.NET 2008. Fig. 3 shows an example of the syntax differences in both languages.

[1] / all this is literal text and will be matched /

```
PERL:     if ($x != $y) {
              $z = $z + 1;
          }

VB.NET:   IF x <> y THEN
              z = z + 1
          END IF
```

Fig. 3 PERL vs. VB.NET

The Software Program Statements that are considered in this paper include simple structures that are commonly used in programming such as: FOR … NEXT; IF … THEN … ELSE, WHILE … END WHILE as well as useful statements such as Addition (x = y + z), Subtraction (x = y - z), Multiplication (x = y * z) and Division (x = y / z).

The following section will explain in detail how Genotypes are converted to either REs or Software Program Statements using XML rules.

3.4 XML Mapping – REs

It is very important for the Genotypes and Phenotypes to have a direct relationship between them, in order for essential characteristics to be preserved from the parents and inherited by the offspring [28]. Koza [17] did not encounter this problem, because the Genotypes were not separated from the Phenotypes in his work, however, representations such as grammatical evolution [20] could be in danger of creating offspring that do not inherit important qualities from their parents, due to a lack of direct mapping between the parent and offspring Phenotypes.

Our Genotype to Phenotype mapping is similar to the work of Withall et al. [28], whereby the Genotypes are used for the genetic manipulation, whereas the Phenotypes are used for the Fitness Test. One of the differences, however, is the length of the Genomes. Withall et al. use fixed-length gene blocks or Genomes, where each block comprises four genes, however, they allow for variable-length Genomes through padding – in cases of shorter program structures or statements, left over genes are ignored. In this research, we only use variable-length Genomes. This is because different REs may contain a varying number of components (i.e. tags, RE structures, keywords etc.). One can create a regular expression that is a centimetre long or one that covers a whole A4 page. Withall et al., however, deal with the evolution of structures, as discussed in section 3.3, which are more rigid when it comes to the number of components they contain.

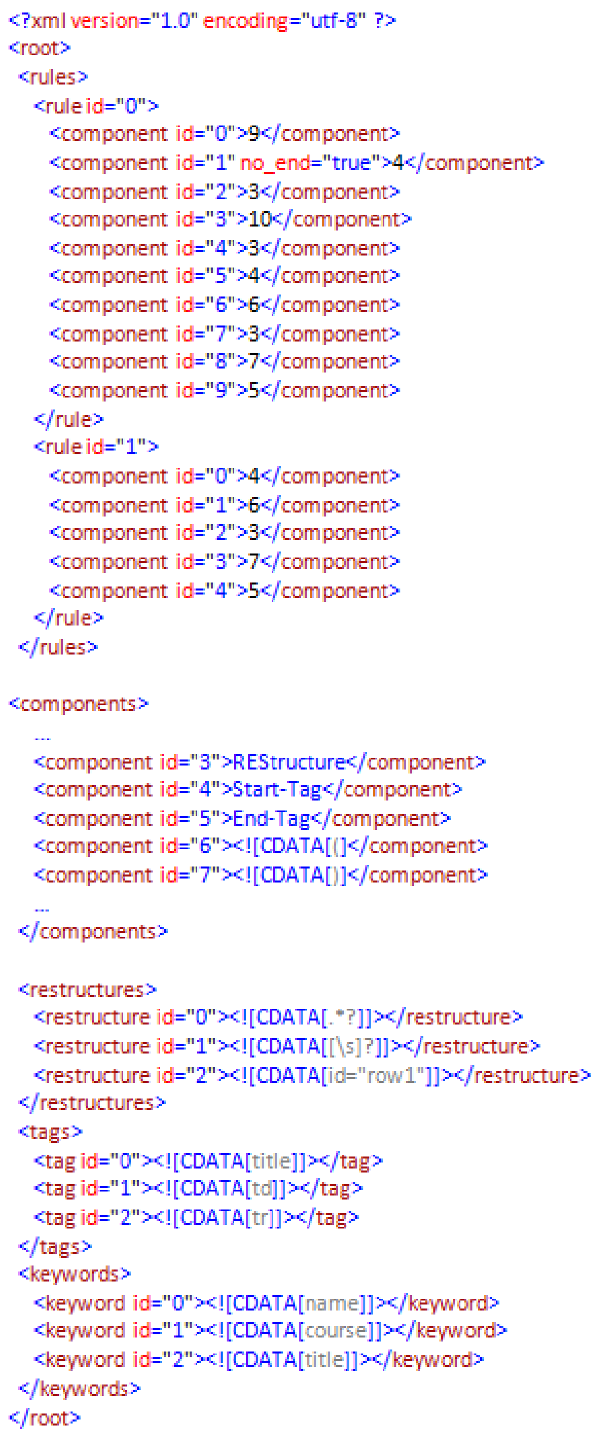
As mentioned previously, a unique attribute of this research is that it achieves the Genotype to Phenotype conversion through XML rules (Fig. 4). This technique has many advantages including: improved readability, compatibility with many programming languages, portability and extendibility (XML is not restricted to a limited set of keywords defined by the proprietary vendors, which aids the process of creating rules of different levels of complexity).

The initial XML rules were created manually after an extensive analysis of a number of web pages. However, in the future, new rules will be able to be added to the XML file automatically (Hinde, Stone and Siau are currently working towards implementing this functionality).

The rest of this section explains the way XML is used to guide the mapping of Genotypes to valid Phenotypes. The Genotype-Phenotype mapping process for the REs domain is shown below:

```
Pseudo-code: Genotype-Phenotype Mapping Process
1) Determine the XML rule to follow
2) Follow the chosen XML rule to the end
3) IF the Genotype has fewer genes than the rule requires
   a) Follow the rule for the number of genes available
   b) Repair outcome to create a valid partial solution.
4) IF the Genotype has enough genes for the XML rule
   a) Follow all the components within the rule
   b) Repair outcome (if necessary) to create a valid
      and complete solution.
5) IF the Genotype has more genes than the rule requires
   a) Follow the same steps as above (4a and 4b)
   b) Ignore the rest of the genes in the Genotype
```

Note that this is not a character by character evolution, because this would increase the search space and dramatically increase the execution time. Instead REs are divided into three collections: HTML tags (e.g. “title”, “tr” etc.), keywords (e.g. “course”, “title” etc.) and RE substructures (e.g. “.*?” or “[\s]?” etc.). Each evolved gene will be translated to an element of one of these collections. Each collection is further divided into a number of components e.g. the Start-Tag component determines an opening tag, the End-Tag determines a closing tag, the Start-REStructure determines an opening RE structure, an RE-Structure determines a normal structure that does not need closing etc.



```xml
<?xml version="1.0" encoding="utf-8" ?>
<root>
 <rules>
   <rule id="0">
     <component id="0">9</component>
     <component id="1" no_end="true">4</component>
     <component id="2">3</component>
     <component id="3">10</component>
     <component id="4">3</component>
     <component id="5">4</component>
     <component id="6">6</component>
     <component id="7">3</component>
     <component id="8">7</component>
     <component id="9">5</component>
   </rule>
   <rule id="1">
     <component id="0">4</component>
     <component id="1">6</component>
     <component id="2">3</component>
     <component id="3">7</component>
     <component id="4">5</component>
   </rule>
 </rules>

<components>
   ...
   <component id="3">REStructure</component>
   <component id="4">Start-Tag</component>
   <component id="5">End-Tag</component>
   <component id="6"><![CDATA[(]</component>
   <component id="7"><![CDATA[)]</component>
   ...
 </components>

 <restructures>
   <restructure id="0"><![CDATA[.*?]]></restructure>
   <restructure id="1"><![CDATA[[\s]?]]></restructure>
   <restructure id="2"><![CDATA[id="row1"]]></restructure>
 </restructures>
 <tags>
   <tag id="0"><![CDATA[title]]></tag>
   <tag id="1"><![CDATA[td]]></tag>
   <tag id="2"><![CDATA[tr]]></tag>
 </tags>
 <keywords>
   <keyword id="0"><![CDATA[name]]></keyword>
   <keyword id="1"><![CDATA[course]]></keyword>
   <keyword id="2"><![CDATA[title]]></keyword>
 </keywords>
</root>
```

Fig. 4 Sample of XML Rules for REs

These components are important for ensuring consistency and accuracy in the Phenotypes created. For example once a Start-Tag has been determined, the system automatically knows that it needs to reuse this tag as an End-Tag when told so by the rules.

We have also introduced some additional components that are not related to the specified collections e.g. the Start-Capture component translates to the symbol '(' and indicates the beginning of a capturing group i.e. the part of the RE, which will capture the part of the results needed; the End-Capture component indicates the end of the capturing group etc. These additional components do not use any genes from the Genotype. They were introduced simply to help the mapping process to translate Genotypes into syntactically correct REs. Fig. 4 shows a partial list of the components needed for the Genotype-Phenotype Mapping process and the order in which they are used within the XML rules.

Each XML rule (Fig. 4) determines the necessary components and the order in which they are to be chosen by the mapping stage in order to create a valid and efficient RE. The first gene in the Genotype is always associated with the RE rule choice. The remaining string of integers in the Genotype maps to the different components within that RE rule.

The modulo function is used for this purpose, e.g. Table 1 shows the Genotype to be translated using the information in Fig. 4. The value of the first gene is 5. This represents the RE rule to be used. In this case, there are two different rules in the XML file, so 5 mod 2 = 1, which means that the second rule (id = 1) is chosen. This rule contains five different components. The first component number is 4 (Table 2). This corresponds to the Start-Tag component (Fig. 4), which means the next gene in the Genotype (gene 7) needs to be mapped to one of the tags in the 'tags' collection. In this case, the collection has three tags, thus 7 mod 3 = 1 gives the 'td' tag. The following component number is 6. This corresponds to the Start-Capture component, which maps to the symbol "(" without using any genes from the Genotype.

The remaining components are dealt with in the same manner (see Table 2). Note that in this example, only the first three genes of the Genotype were needed; the remaining two are simply ignored.

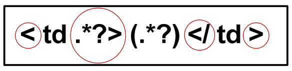
Table 1: Genotype

5	7	15	22	43

Table 2: Genotype to RE Mapping

Component No.	*Component*	*Gene Used*	*Modulo*	*Translation*
-	-	5	1	Rule 2
4	Start-Tag	7	1	td

6	Start-Capture	-	-	(
3	RE-Structure	15	0	.*?
7	End-Capture	-	-	)
5	End-Tag	-	-	td



<td.*?>(.*?)</td>

Fig. 5 Phenotype

Once all the required genes have been decoded, the resulting RE is repaired to ensure its validity and efficiency. Fig. 5 shows the complete RE (Phenotype) for the above example. The additional symbols added by the repairing function are shown encircled.

The repairing function is an independent function which scrutinises the Phenotype created in order to guarantee the validity of the RE[2]. This function is in charge of tasks like: closing opening tags, closing opening parentheses, adding/removing RE structures in cases when there are fewer genes in the Genotype than required by the XML rule, adding variable declarations at the beginning of the program, adding 'footer' information where necessary e.g. when returning the RE to a calling function etc. All this is achieved through the use of a STACK programming structure, which works in a LIFO (Last In First Out) manner. This is particularly helpful when closing nested tags and RE structures, for example:

Tag1 + Tag2 + RE-Structure + *C-RE* + *C-Tag2* + *C-Tag1*

Above, C stands for Closing. The *Italic* part shows the part of the RE added by the repairing function.

The XML rules can also determine whether or not a closing tag is actually needed. This is useful in cases where the inclusion of a closing tag results in different results being extracted to the ones needed. For example, the RE: <tr[\s]?id="row1".*?><td.*?>.*?</td> contains two opening tags ('tr' and 'td'), however, only the 'td' tag needs to be closed for this to work as intended. This is achieved by including no_end="true" in the rule (Fig. 4).

3.5 XML Mapping – Software Program Statements

This section discusses the changes that needed to be made to the system for it to work with Software Program Statements instead of REs. The areas changed were: the XML rules and the repairing function. The following explains the changes involved in order to show the minimum effort required to adapt our system to such an entirely different domain.

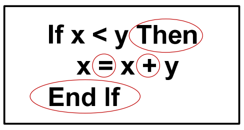
Fig. 7 shows a sample of the XML rules and components needed to guide the Genotype-Phenotype stage of the genetic evolution of software program statements. When compared to the Genotype-Phenotype Mapping system for REs (Fig. 4), it is clear that the overall logic remains the same, with each XML rule using the available collections (in this case: 'variables' and 'comparisons') to guide the system through the different components needed for each rule. Identically, there are a number of components that have been introduced to simplify and ensure the consistency of the mapping process for Software Program Statements, but which do not use any genes from the Genotype since they do not need to be evolved. Such components include: the 'Assign (=) operator, the 'Addition' (+) operator, etc.

Table 3: Genotype

10	27	7	13	19	9	63	4

Table 4: Genotype to Software Statement Mapping

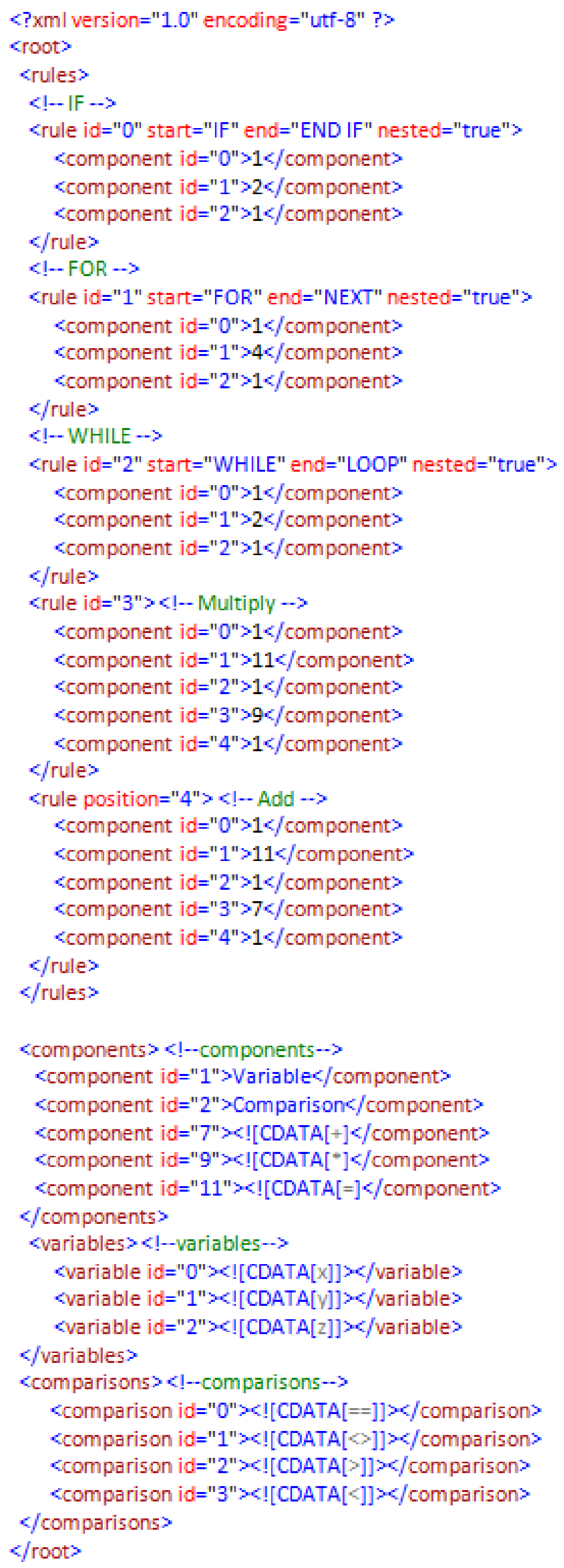
Component No.	*Component*	*Gene Used*	*Modulo*	*Translation*
-	-	10	0	If
1	Variable	27	0	x
2	Comparison	7	3	<
1	Variable	13	1	y
-	-	19	4	Add
1	Variable	9	0	x
11	Assign	-	-	=
1	Variable	63	0	x
7	Addition	-	-	+
1	Variable	4	1	y



If x < y Then
x = x + y
End If

Fig. 6 Phenotype

[2] Many HTML tags come in pairs. The system ensures that the same closing tag is chosen based on the opening tag evolved, and the nesting of the tags in the RE hierarchy is preserved.



```
<?xml version="1.0" encoding="utf-8" ?>
<root>
 <rules>
  <!-- IF -->
  <rule id="0" start="IF" end="END IF" nested="true">
     <component id="0">1</component>
     <component id="1">2</component>
     <component id="2">1</component>
  </rule>
  <!-- FOR -->
  <rule id="1" start="FOR" end="NEXT" nested="true">
     <component id="0">1</component>
     <component id="1">4</component>
     <component id="2">1</component>
  </rule>
  <!-- WHILE -->
  <rule id="2" start="WHILE" end="LOOP" nested="true">
     <component id="0">1</component>
     <component id="1">2</component>
     <component id="2">1</component>
  </rule>
  <rule id="3"> <!-- Multiply -->
     <component id="0">1</component>
     <component id="1">11</component>
     <component id="2">1</component>
     <component id="3">9</component>
     <component id="4">1</component>
  </rule>
  <rule position="4"> <!-- Add -->
     <component id="0">1</component>
     <component id="1">11</component>
     <component id="2">1</component>
     <component id="3">7</component>
     <component id="4">1</component>
  </rule>
 </rules>

 <components> <!--components-->
   <component id="1">Variable</component>
   <component id="2">Comparison</component>
   <component id="7"><![CDATA[+]</component>
   <component id="9"><![CDATA[*]</component>
   <component id="11"><![CDATA[=]</component>
 </components>
  <variables> <!--variables-->
     <variable id="0"><![CDATA[x]]></variable>
     <variable id="1"><![CDATA[y]]></variable>
     <variable id="2"><![CDATA[z]]></variable>
 </variables>
 <comparisons> <!--comparisons-->
     <comparison id="0"><![CDATA[==]]></comparison>
     <comparison id="1"><![CDATA[<>]]></comparison>
     <comparison id="2"><![CDATA[>]]></comparison>
     <comparison id="3"><![CDATA[<]]></comparison>
 </comparisons>
</root>
```

Fig. 7 Sample of XML Rules for Software Statements

For example, Table 3 shows the Genotype to be translated using the information in Fig. 7. The value of the first gene is 10. This represents the statement type to be used. In this case, there are five different rules (statements) in the XML file, so 10 mod 5 = 0 means that the statement is an 'If'. This statement contains three different components each requiring the use of a gene to choose from the 'variables' and 'comparisons' collections. The 'variables' collection has three elements, whereas the 'comparisons' has four, therefore, 27 mod 3 = 0, 7 mod 4 = 3 and 13 mod 3 = 1 give elements 'x', '<' and 'y' respectively.

This is the point where the logic in the Genotype-Phenotype mapping of REs changes slightly. In this case, the system knows that although the first statement has been translated in full, the mapping cannot end here. This is because, the XML rule for the IF statement includes an extra attribute: *nested* = *"true"* (Fig. 7) which indicates that the IF statement expects another statement inside. The following gene (gene 19) in the Genotype is therefore used to determine the next statement type. Therefore, 19 mod 5 = 4 means that the next statement is 'Add'. The remaining components are dealt with in the same manner as previously (see Table 4).

Once all the required genes have been decoded, the resulting Phenotype is repaired to ensure it is syntactically correct. Fig. 6 shows the complete software structure (Phenotype) for the above example. The additional symbols and programming keywords added by the repairing function are shown encircled. The 'Assign' and 'Addition' operators are also dealt with in the repairing function, as these are static attributes associated with the Add statement, thus they do not need to be evolved. The repairing function retains the same STACK programming structure and responsibilities as in the REs domain.

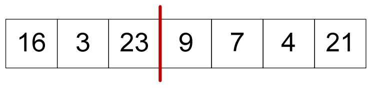
A difference noticed between the two domains is that whilst each rule in Fig. 7 belongs to a different software statement or structure, all of the rules in Fig. 4 are used for the extraction of the same piece of information from the web (the course title in this example). This is because REs are very diverse and as such an unlimited number of REs can be written to extract the same piece of information from a web page. This means that in relation to updating the XML rules once written, the Software Program Statements would require much less attention than REs because these are more rigid structures that need a certain number of components, in a particular order, at all times. For example, the Add statement mentioned above may sum up more than two variables, however, there will always be need for one variable to which this sum is assigned, one 'Assign' operator and one or more 'Addition' operators.

4. Discussion

There is a significant difference between the two domains chosen for this paper. REs are diverse and sometimes unpredictable, whereas Software Program Statements are structured and rigid. They do not have any components in common. It is in fact difficult to find any similarities between them.

One main difference between the two domains, in relation to their GP representation, is that each RE is represented as a linear, string of integers, where all the integers (genes) are part of the same Genome. A Software Program Statement, on the other hand, can be represented as either a string of integers or as a set of strings of integers. The latter is the case with program structures such as: IF ... ELSE ... THEN, or FOR ... NEXT etc. which incorporate other independent software statements or structures within them. Fig. 3 showed an example of an ‘Add’ statement being nested within the IF structure.

Withall et al. [28] deals with these cases by having separate Genomes for each individual statement. Updating our system to deal with multiple Genomes, would potentially require a considerable change, thus, all the Genomes were instead joined to form one single Genome. Similarly to the example discussed in section 3.5, Fig. 8 shows an example of the Genotype-Phenotype mapping process treating two Genomes (separated with the vertical red line) as one and mapping them to the FOR ... Add ... NEXT structure below. This example works with the data shown in Fig. 7.



```
FOR x = 0 To z
    y = y + x
NEXT
```

Fig. 8 Single Genome Mapping

The only change required to map joined Genomes to a valid Phenotype was to alter the system to recognise the end of one statement and the beginning of the other. In the above example, the system knows that FOR is a nested structure; additionally the XML rule tells the system that the FOR structure needs only two variables, thus the system deduces that the fourth gene (gene 9) must belong to a different statement or structure nested within the FOR loop. Gene 9 corresponds to the ‘Add’ statement, thus the genes following this one will be mapped according to the XML rule for ‘Add’. The repairing function is executed at the end to tidy up the solution by closing the FOR structure.

The above change took the longest to complete – nearly three hours – as it involved altering message calls within two different classes in our Object Oriented System. The changes stated in section 3.5 were all fairly simple to implement and took under 30 minutes to change. Thus overall, this has been a 3.5 hour effort, which considering the significant differences in both domains, we consider this to be a minimal effort.

Results from the execution of our full genetic system in the evolution of REs for the extraction of course names from web pages can be found in [31]. This system would not have to change to work with any other domain that requires the use of REs. However, running the full system on the Software Program Statements domain, or other RE-unrelated domains, would require an additional change – a different Fitness test.

5. Conclusions

This paper has discussed an innovative approach to mapping Genotypes to Phenotypes through XML rules. The different steps involved in the process are explained through examples. Two entirely different domains were considered – REs and Software Program Statements - in order to show the amount of effort and time required to adapt the original system to work with a new domain.

Results show that the effort involved in the alterations was minimal, with all the changes taking under 3.5 hours. More time, however, needs to be spent to optimise the translation of constant variables.

The next stage of the research will concentrate on using our Genotype-Phenotype mapping process together with the rest of the GP system to evolve REs for the extraction of other training course details such as prices, dates and locations. Analysis of some preliminary web pages has indicated a partial dependence amongst this data, which will need to be reflected in the Fitness test produced.

Acknowledgments

We would like to thank the whole team at ATM for their support and help. Also, thank you to ATM, the Centre for Innovative and Collaborative Engineering (CICE) and Loughborough University for funding our work. We would also like to thank Daniel Sills for his help with some technical .NET concepts.

References

[1] Back, T., Hammel, U. & Schwefel, H-P. 1997. Evolutionary Computation: Comments on the History and Current State. IEEE Transactions on Evolutionary Computation.

[2] Banzhaf, W. 1994. Genotype-Phenotype-Mapping and Neutral Variation – A case study in Genetic Programming. Proceedings of the International Conference on Evolutionary Computation. Springer-Verlag, 322-332.

[3] Barrero, D., Camacho, D. & R-Moreno, M. 2009. Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions. Data Mining and Multi-agent Integration. ISBN 978-1-4419-0523-9, Springer-Verlag US, 143.

[4] Clarke, M., Hinde, C.J., Whithall, M.S., Jackson, T.W., Phillips, I.W., Brown, S & Watson, R. 2009. Allocating Railway Platforms using a Genetic Algorithm. Research and Development in Intelligent Systems XXVI, Springer London, 421-434.

[5] Conrad, E. 2007. Detecting Spam with Genetic Regular Expressions. SANS Institute Reading Room. Available online at: http://www.giac.org/certified_professionals/practicals/GCIA/00793.php

[6] De Kunder, M. 2010. The size of the World Wide Web (Tilburg University). Retrieved February 23rd, 2010 from http://www.worldwidewebsize.com

[7] Dyer, J. & Bentley, P. 2002. PLANTWORLD: Population Dynamics in Contrasting Environments. In Cantu-Paz E., GECCO, 122-129.

[8] Fogel, D.B. 1994. An Introduction to Simulated Evolutionary Optimisation. IEEE Transactions on Neural Networks.

[9] Fogel, D.B. 1998. The Fossil Record. Fogel, D.B., Ed., IEEE Press

[10] Friedl, J. 2006. Mastering Regular Expressions, Third Edition. O'Reilly & Associates (Aug 2006)

[11] Goyvaerts, J. RegexBuddy. http://www.regular-expressions.info/regexbuddy.html

[12] Hsu, P-H. 2007. Feature extraction of hyperspectral images using wavelet and matching pursuit. ISPRS Journal of Photogrammetry and Remote Sensing. Elsevier Science, Amsterdam, vol. 62 (2), 78-92.

[13] Jackson, D. 2005. Evolving Defence Strategies by Genetic Programming. In Lecture Notes in Computer Science. Springer Berlin, Vol. 3447, 281-290.

[14] Keaveney, D. & O'Riordan, C. 2009. Evolving Robust Strategies for an Abstract Real-time Strategy Game. Proceedings of the 5th International Conference on Computational Intelligence and Games. 371-378.

[15] Keller, R.E. & Banzhaf, W. 1996. Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes. Proceedings of the First Annual Conference on Genetic Programming, California. 116-122.

[16] Klank, U., Padoy, N., Feussner, H. and Navab, N. 2008. Automatic feature generation in endoscopic images. International Journal of Computer Assisted Radiology and Surgery. Springer, 3, 331-339

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

[18] Langdon, W., Poli, R., McPhee, N. & Koza, J.R. 2008. Genetic Programming: An Introduction and Tutorial with a Survey of Techniques and Applications. In Studies in Computational Intelligence. Springer, Berlin, vol. 115, 927-1028.

[19] Moore, J.P. 2000. Exploring and Exploiting Models of the Fitness Landscape: A Case against Evolutionary Optimization. PhD Thesis, University of Plymouth.

[20] O'Neill, M., Brabazon T., Ryan, C. & Collins J.J. 2001. Developing a Market Timing System using Grammatical Evolution. Proceedings of GECCO.

[21] Poli, R., Langdon, W. & McPhee. 2008. N. A Field Guide to Genetic Programming. Published via http://lulu.com (With contributions from J. R. Koza)

[22] Regular-Expressions.info (Last Update: Jun 2009). http://www.regular-expressions.info/

[23] Rothlauf, F. 2006. Representations for Genetic and Evolutionary Algorithms. Springer-Verlag New York.

[24] Sells, C. 2009. RegexDesigner.NET http://regexdesigner-net.findmysoft.com/

[25] Sun Microsystems. (Last updated: Feb 2008). Lesson: Regular Expressions. http://java.sun.com/docs/books/tutorial/essential/regex/index.html

[26] Thompson, K. 1968. Programming techniques: Regular expression search algorithm. Commun. ACM, Vol 11(6), 419-422

[27] Whigham, P.A. 1995. Grammatically-based Genetic Programming. Workshop on Genetic Programming.

[28] Withall, M.S., Hinde, C.J. & Stone, R.G. 2008. An improved representation for evolving programs. Journal of Genetic Programming and Evolvable Machines. Springer Netherlands, vol. 10(1), 37-70.

[29] Xhemali, D., Hinde, C.J. and Stone, R.G. 2007. Embarking on a Web Information Extraction Project. UKCI Conference on Computational Intelligence (London, UK, Jul 02-04, 2007)

[30] Xhemali, D., Hinde, C.J. & Stone, R.G. 2009. Naive Bayes vs. Decision Trees vs. Neural Networks in the Classification of Training Web Pages. International Journal of Computer Science Issues, vol. 4(1), 16-23.

[31] Xhemali, D., Hinde, C.J. & Stone, R.G. 2010. Genetic Evolution of Regular Expressions for the Automated Extraction of Course Names from the Web. Internal Report. Loughborough University, UK.

Daniela Xhemali is an Engineering Doctorate (EngD) student at Loughborough University, UK. She received a First Class (Honours) BSc in Software Engineering from Sheffield Hallam University in 2005 and an MSc with Distinction in Engineering Innovation and Management from Loughborough University in 2008. Daniela Xhemali has also worked in industry for two years as a Software Engineer, programming multi-user, object oriented applications, with large database backend. Her current research focuses on the use of Genetic Programming principles for the extraction of web information.

Prof. Christopher J. Hinde is the Programme Director of the Computer Science & Artificial Intelligence group as well as the Programme Director of the Computer Science & E-business group at Loughborough University. Prof. Hinde is also the leader of the Intelligent and Interactive Systems Research division. His research interests include: Artificial intelligence, fuzzy reasoning, logic programming, natural language processing, neural nets etc.

Dr. Roger G. Stone is DANS Coordinator and the Quality Manager at Loughborough University. Dr. Stone is also a member of the Interdisciplinary Computing Research Division. His research interests include: Web programming, web accessibility, program specification techniques, software engineering tools, compiling etc.