Automatic Generation of High Level Functions using Evolutionary Algorithms

Michael O'Neill & Conor Ryan
Dept. Of Computer Science And Information Systems
University of Limerick
Ireland
{Michael.ONeill|Conor.Ryan}@ul.ie

Abstract

Evolutionary Algorithms, based upon the Darwinian principle of Natural Selection, have been applied with varying degrees of success to a broad range of problem domains, including the automatic induction of computer programs. This paper describes Grammatical Evolution (GE), an evolutionary algorithm approach to automatic programming, which employs variable length, binary chromosomes. The binary chromosome is used in a genotype to phenotype mapping process to select production rules of a Backus Naur Form (BNF) grammar definition in order to generate syntactically correct programs in an arbitrary language specified by the BNF definition.

Part of the power of GE is that it is closer to natural DNA than other Evolutionary Algorithms, and thus can benefit from natural phenomena such as a separation of search and solution spaces through the genotype to phenotype mapping, and a genetic code degeneracy which can give rise to silent mutations (mutations that have no effect on the phenotype).

1 Introduction

Grammatical Evolution (GE) is a grammar based, linear genome system which is capable of generating programs or expressions in any language. Rather than the functions and terminals associated with GP [3], GE takes a BNF specification of a language, or subset thereof, from which it can subsequently generate compilable code. The BNF is used to build a program by applying production rules to elements of the non-terminal set of the BNF definition, in a mapping process to generate the output code from a simple binary string. The evolutionary process is thus applied to the binary strings that are the individuals in the population of the evolutionary algorithm. During the development of GE we have attempted to harness some of the features of the genetic machinery of living organisms which are theorised to have an impact on the phenomenon of evolution [1].

GE has been successfully applied to a number of diverse problem domains such as symbolic regression [8], finding trigonometric identities [9], symbolic integration [10], and the Santa Fe trail [5]. The results compared favorably with systems such as GP, and have been shown to outperform GP [5]. A description of GE follows.

2 Backus Naur Form

Backus Naur Form (BNF) is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of `terminals`, which are items that can appear in the language, and `non-terminals`, which can be expanded into one or more terminals and non-terminals. A grammar can be represented by the tuple, $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. For example, below is the BNF used for the Santa Fe ant trail problem [5], where

$$N = \{code, line, expr, if - statement, op, if - true, if - false\}$$

$$T = \{left(), right(), move(), food\_ahead(), else, if, \{, \}, (,), ;\}$$

$$S =< code >$$

And P can be represented as:

```
(1) <code> : : =  <line>              (A)
                 |<code><line>        (B)

(2) <line> : : =  <expr>

(3) <expr> : : =  <if-statement>      (A)
                 |<op>                (B)

(4) <if-statement> : : = if(food_ahead()){<expr>}else{<expr>}

(5) <op> : : =    left();             (A)
                | right();            (B)
                | move();             (C)
```

Unlike a Koza-style approach, there is no distinction made at this stage between what he describes as functions (operators in this sense) and terminals, however, this distinction is more of an implementation detail than a design issue. In GE the BNF definition is used to describe the output language produced by the system, that is, the compilable code produced will consist of elements of the terminal set T. As the BNF is a plug-in component of the system it means that GE can produce code in any language that can be specified in the form of a BNF definition giving the system a unique flexibility.

3 The Biological Approach

The GE system is inspired largely by the biological process of generating a protein from the genetic material of an organism. Proteins are fundamental in the proper development and operation of living organisms, and are responsible for traits such as eye colour, and height [1].

The genetic material, usually called DNA, contains the information required to produce specific proteins at different points along the molecule. For simplicity, consider DNA to be a string of building blocks called nucleotides, of which there are four, named A, T, G, and C, for Adenine, Tyrosine, Guanine, and Cytosine respectively. Groups of three nucleotides, called a codon, are used to specify the building blocks of proteins. These protein building blocks are known as amino acids, and the sequence of these amino acids in the protein is determined by the sequence of codons on the DNA. The sequence of amino acids is very important as it plays a large part in determining the final three dimensional structure of the protein, which in turn has a role to play in determining its functional properties.

In order to generate a protein from the sequence of nucleotides in the DNA, the nucleotide sequence is firstly transcribed into a slightly different format, that being a sequence of elements on a molecule known as RNA. The RNA molecule contains elements that are a slightly modified form of the nucleotides that are contained within the DNA molecule. Codons within the RNA molecule are then translated to determine the sequence of amino acids that are contained within the protein molecule.

The result of the expression of the genetic material as proteins in conjunction with environmental factors is termed it's phenotype. In GE the phenotype is a running computer program. The process of generating the phenotype from the genetic material (the genotype) is termed a genotype-phenotype mapping process, which is unlike the standard method of generating a solution (a program in the case of GE) directly from an individual in an evolutionary algorithm by explicitly encoding the solution within the genetic material. Instead, a many-to-one mapping process is employed within which the real power of the GE system lies.

Figure 1 outlines the mapping process employed in both GE, and biological organisms.

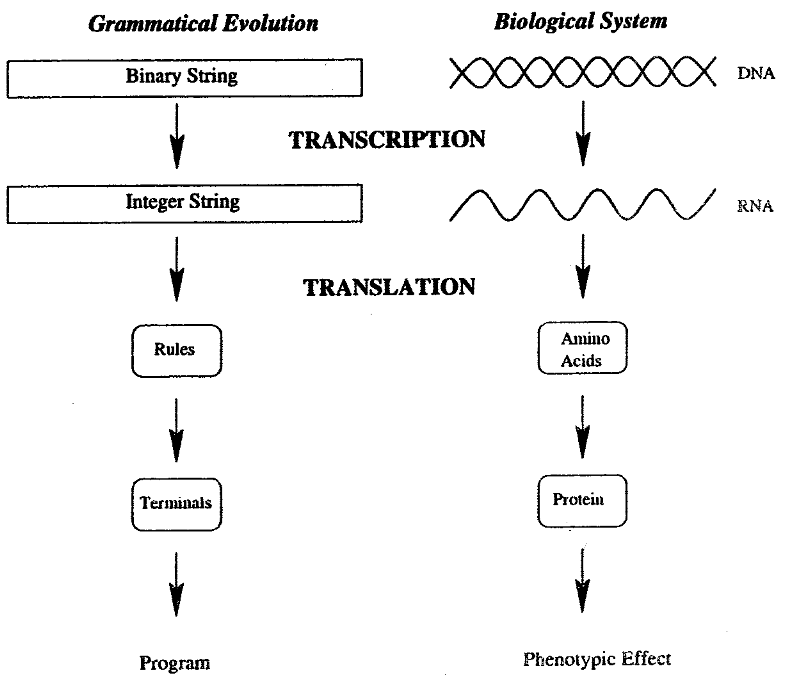


Figure 1: The Grammatical Evolution System and a Biological System

4 Grammatical Evolution

When tackling any problem with GE, a suitable BNF definition must first be decided upon. The BNF can be either the specification of an entire language, or perhaps more usefully, a subset of a language geared towards the problem at hand.

4.1 The Mapping Process

The genotype is then used to map the start symbol onto terminals by reading codons of 8 bits to generate a corresponding integer value, from which an appropriate production rule is selected. A rule is selected by using the following,

$$(Integer\ Codon\ Value)\ MOD$$

$$(Number\ of\ Production\ Rules\ for\ the\ current\ non-terminal)$$

Considering the following rule,

```
(5) <op> :: =   left();          (A)
              | right();         (B)
              | move();          (C)
```

i.e. given the non-terminal *op* there are three production rules to select from. If we assume the codon being read produces the integer 6, then $6\ MOD\ 3\ =\ 0$ would select rule $(A)\ left()$. Each time a production rule has to be selected to map from a non-terminal, another codon is read, and in this way, the system traverses the genome.

During the genotype to phenotype mapping process it is possible for individuals to run out of codons, and in this case we wrap the individual, and reuse the codons. This is quite an unusual approach in EA's, as it is entirely possible for certain codons to be used two or more times. This technique of wrapping the individual draws inspiration from the gene overlapping phenomenon which has been observed in many organisms in nature [1].

In GE, each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is being applied, it may result in the selection of a different production rule.

What is crucial, however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated, this is because the same choices are made each time. It is possible that an incomplete mapping could occur, even after wrapping, and in this event the individual in question is given the lowest fitness value possible, then the selection and replacement mechanisms should operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules over and over. For example, given an individual with three codons, if the first codon specified rule B from below,

```
(1) <code> :: =   <line>          (A)
                 |<code><line>    (B)
```

and the second, and third also specified this same rule, even after wrapping the mapping process would be incomplete and would carry on indefinitely unless stopped. To reduce the number of invalid individuals being passed from generation to generation, a Steady State replacement mechanism is employed, which in itself is a more biologically plausible than the standard generational replacement mechanisms employed by many other evolutionary algorithms. A result of the Steady State method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals.

4.2 Example Genotype to Phenotype Mapping

Using the grammar as given in Section 2, we shall show how the following individual is mapped onto the output language, i.e. elements of the grammars terminal set.

Consider an individual made up of the following codons (expressed in decimal for clarity) :

220	10	17	3	109	215	104	30

The mapping process begins from the Start symbol, in this case `<code>`, which has two productions to choose from. To select the rule to use the first 8 bit codon on the individual is read and converted to an integer value, in this case 220. Using the formula given in Section 4.1 we obtain the rule to be applied to this non-terminal, i.e. 220 MOD 2 = 0, selecting rule (A) `<line>`. `<code>` has now been replaced with `<line>`. The mapping process continues by selecting productions rules for the leftmost non-terminal by reading the next codon on the individual being mapped. The next non-terminal in this case has to be `<line>` which has no choices, and therefore no codon value is read, instead it is replaced by `<expr>`. The `<expr>` non-terminal however has two choices and so the next codon value , 10, is read in order to select the next production rule. This gives 10 MOD 2 = 0, i.e. rule (A) `<expr>` is replaced with `<if-statement>`. There are no choices for the `<if-statement>` non-terminal and so it is replaced directly with `if(food_ahead()){<expr>}else{<expr>}`. The next leftmost non-terminal is now `<expr>`, which is replaced with `<op>`, according to 17 MOD 2 = 1, which selects rule (B). The incomplete individual now has the following form,

```
if(food_ahead()){<op>}else{<expr>}.
```

The next non-terminal to be mapped is `<op>` which has three choices, and the next codon has the integer value 3 which gives 3 MOD 3 = 0, this selects rule (A). The individual has become,

```
if(food_ahead()){left();}else{<expr>}.
```

The `<expr>` non-terminal is now replaced by reading the next codon value, 109, and getting modulus two of it as there are two choices to be made, i.e. 109 MOD 2 = 1, rule (B). The individual now takes the form,

```
if(food_ahead()){left();}else{<op>}.
```

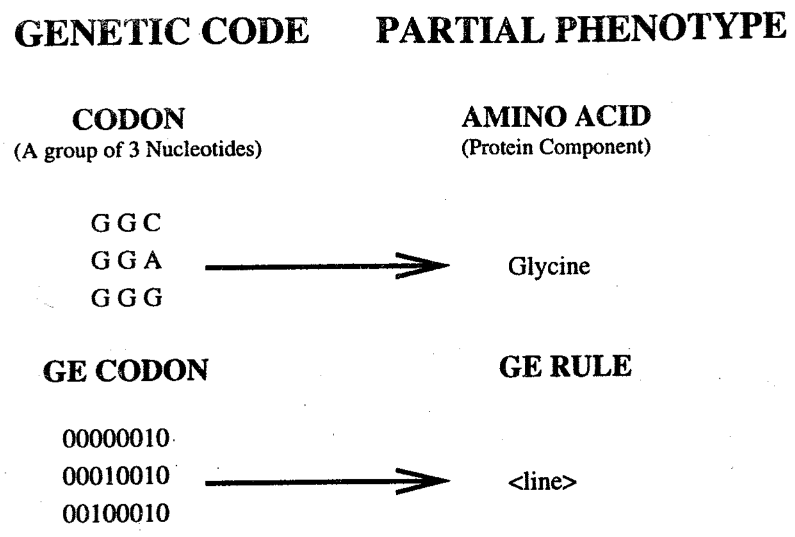
Again, the non-terminal `<op>` has three choices, and so by reading the next codon value we get the following, 215 MOD 3 = 2, rule (C). This says that `<expr>` is replaced with `move();`. There now exists a completely mapped individual of the form,

```
if(food_ahead()){left();}else{move();}.
```

The extra codons at the end of the individual are simply ignored in the mapping process.

4.3 Genetic Code Degeneracy

Given an 8 bit binary number, each codon in GE can represent 256 distinct integer values. However, many of these integer values can represent the same production rule, taking production rule 5 as an example, if the current codon value was 6, then 6 MOD 3 = 0 would select rule (A) $left()$ as shown above. The same rule would be chosen if the codon value was 3, 9, 12, etc.



For Rule (1) in the example BNF, where
<code> :: = <line> (0)
| <code><line> (1)
i.e. (GE Codon Integer Value) MOD 2 = Rule Number

Figure 2: Genetic Code Degeneracy

A similar phenomenon can be observed in the genetic code of biological organisms, referred to as *Degenerate Genetic Code* [1]. There are 4^3, i.e. 64, unique combinations of nucleotides in a codon, 61 of these coding for a specific amino acid, the other three are special codons which delimit the end of a gene on the DNA. On average, there are three codons for every amino acid, that is more than one codon can represent the same amino acid, and it has been observed that the first two nucleotides in the codon are often sufficient to specify a particular amino acid, and so the value of the nucleotide at the third position is often irrelevant. Code degeneracy has interesting implications when it comes to mutation effects. A mutation at the third codon position can often produce what is called a *silent mutation*, meaning that the amino acid specified will be the same as the one before the mutation event, due to the flexibility at the third codon position. With respect to GE this means that subtle changes in the search space (genotype) may have no effect on the solution space (phenotype), which could result in the maintenance of genotypic diversity throughout a run of the system, and the preservation of valid individuals. Figure 2 shows that in the genetic code of biological organisms the nucleotide at position three of the codon is independent of the amino acid produced (Valine). Similarly with GE, it can be seen in the given example that a single bit mutation has no effect on the rule used in this case i.e. $2\ MOD\ 2 = 18\ MOD\ 2 = 34\ MOD\ 2 = 66\ MOD\ 2 = 0$, the 0th rule *line*.

Kimura's neutral theory of evolution [2] states that it is these silent mutations that are responsible for the genetic diversity which has been observed in natural populations, a phenomenon which has been exhibited within GE.

As the population being evolved comprises simple binary strings, we do not have to employ any special crossover or mutation operators, and as such an unconstrained search is performed on these strings due to the genotype to phenotype mapping process which will generate syn-

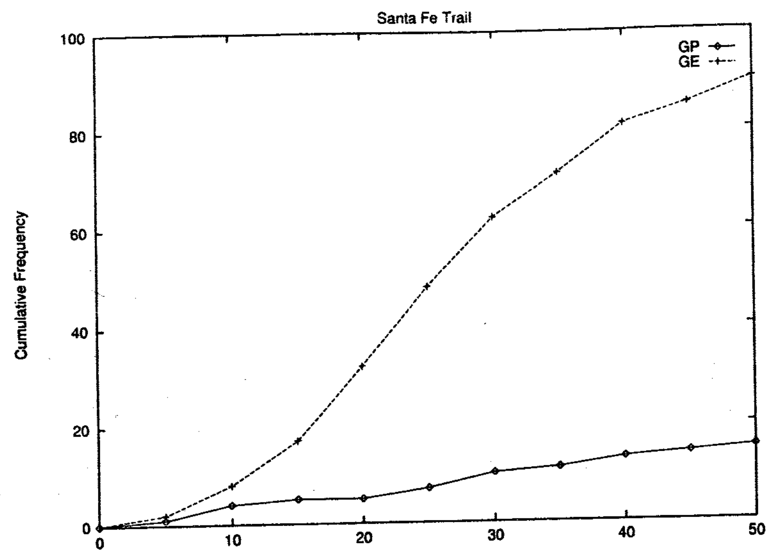


Figure 3: A cumulative frequency measure of GE in comparison to GP.

```
if(food_ahead())
  { move(); }
else
  { move(); }
if(food_ahead())
  { left(); }
else
  { left(); }
if(food_ahead())
  { left(); }
else
  { right(); }
if(food_ahead())
  { right(); }
else
  { right(); }
if(food_ahead())
  { move(); }
else
  { left();}
```

6 Conclusions

This paper has served to describe the biological principles employed by the Grammatical Evolution system. Future papers [6] [7] will show that the benefits of a complex mapping process show that the one-to-one genotype to phenotype mapping that is so prevalent in other evolutionary algorithms is not necessarily a good idea, and that by incorporating this and other biological principles taken from natural genetics that the performance of this system and perhaps other evolutionary algorithm techniques can be improved.

References

[1] Elseth Gerald D., Baumgardner Kandy D. 1995. Principles of Modern Genetics. *West Publishing Company.*

[2] Kimura, M. 1983. The Neutral Theory of Molecular Evolution. Cambridge University Press.

[3] Koza, J. 1992. *Genetic Programming.* MIT Press.

[4] Langdon, W. & Poli, R. Why Ant's Are Hard. In *Proceedings of Genetic Programming 1998*, pages 193-201 .

[5] O'Neill M., Ryan C. 1999. Evolving Multi-line Compilable C Programs. In *Proceedings of the Second European Workshop on Genetic Programming 1999.*

[6] O'Neill M., Ryan C. 1999. Under the Hood of Grammatical Evolution. In *Proceedings of the Genetic & Evolutionary Computation Conference 1999.*

[7] O'Neill M., Ryan C. 1999. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. Submitted to the 5th European Conference on Artificial Life 1999.

[8] Ryan C., Collins J.J., O'Neill M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming*, pages 83-95. Springer-Verlag.

[9] Ryan C., O'Neill M., Collins J.J. 1998. Grammatical Evolution: Solving Trigonometric Identities. In *Proceedings of Mendel '98: 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks and Rough Sets*, pages 111-119.

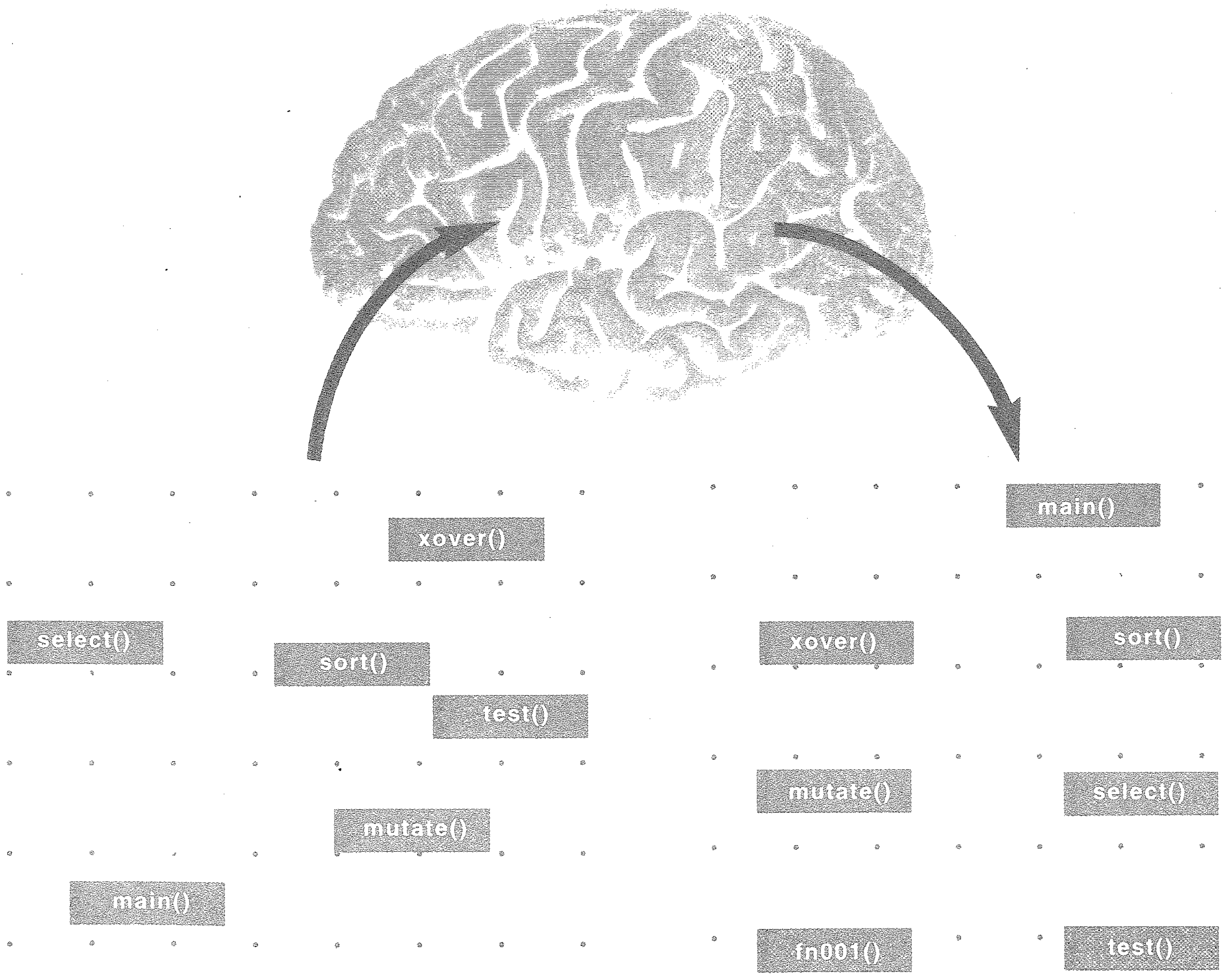
[10] Ryan C., O'Neill M. Grammatical Evolution: A Steady State Approach. In *Late Breaking Papers, Genetic Programming 1998*, pages 180-185.

[illegible]

SCASE'99

SOFT COMPUTING APPLIED TO SOFTWARE ENGINEERING

UNIVERSITY OF LIMERICK, IRELAND
APRIL 1999



edited by Conor Ryan and Jim Buckl

SCASE'99

Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering

Limerick, Ireland.
12th-14th April, 1999

Editors:

C. Ryan and J. Buckley

ISBN: 1-874653-52-6
Limerick University Press.

SCARE
Soft Computing and Re-Engineering Group.
University of Limerick, Ireland.
Funding by Forbairt.