Proceedings of the 10th WSEAS International Conference on SYSTEMS, Vouliagmeni, Athens, Greece, July 10-12, 2006 (pp406-411)

jGE - A Java implementation of Grammatical Evolution

LOUKAS GEORGIOU and WILLIAM J. TEAHAN
School of Informatics, University of Wales, Bangor, U.K.

Abstract: Grammatical Evolution (GE) is a novel evolutionary algorithm which uses an arbitrary variable-length binary string to govern which production rule of a Backus Naur Form grammar will be used in a genotype-to-phenotype mapping process. This paper introduces the Java GE project (jGE), which is an implementation of GE in the Java language. The main difference between jGE and libGE, a public domain implementation of GE in C++, is that jGE incorporates the functionality of libGE as a component and provides implementation of the Search Engine as well as the Evaluator. The main idea behind the jGE Library (it can be downloaded at [16]) is to create a framework for evolutionary algorithms which can be extended to any specific implementation such as Genetic Algorithms, Genetic Programming and Grammatical Evolution.

Key-Words: Grammatical Evolution, Genetic Algorithms, Evolutionary Computation, Agents, jGE, libGE, GP

1 Grammatical Evolution

1.1 Motivation

The main goal of the Java GE (jGE) project at Bangor is the implementation of an Evolutionary Algorithms (EA) framework which will facilitate further research into Evolutionary Algorithms (and especially Grammatical Evolution). Grammatical Evolution [8] was chosen as the main Evolutionary Algorithm of the jGE Project because it facilitates, due the use of a BNF Grammar, the evolution of arbitrary structures and programming languages.

Other objectives of the jGE Library (and some further reasons as to why Java was chosen as the implementation language) are as follows:

- An open and extendable framework for the experimentation with EAs;
- The creation of an Agent-Oriented Evolutionary System (an agent-based framework);
- Bootstrap project for further research on the application of the principles of the Evolutionary Synthesis theory [5] in machines;
- Integration and interoperability with other projects such as evolutionary algorithms with knowledge sharing [15];
- Integration with other open source and free Java projects like Robocode (e.g. evolution of simulated robots using GE).

1.2 The Grammatical Evolution system

Grammatical Evolution [9] is an evolutionary algorithm that can evolve complete programs in an arbitrary language using a variable-length binary string. The binary string (genome) determines which production rules in a Backus Naur Form (BNF) grammar definition are used in a genotype-to-phenotype mapping process to generate a program.

O'Neill and Ryan [9] take inspiration from nature and claim that Grammatical Evolution embraces the developmental approach and draws upon principles that allow an abstract representation of a program to be evolved. This abstraction enables GE to do the following things: it separates out the search and solution spaces; it allows evolution of programs in an arbitrary language; it enables the existence of degenerate genetic code; and it adopts a wrapping operation that allows the reuse of the genetic material.

According to O'Neill and Ryan [9], Grammatical Evolution is based on the principles of the fields of Evolutionary Automatic Programming, Molecular Biology, and Grammars. Although Grammatical Evolution is a form of Genetic Programming, it differs from traditional GP in three ways [9]: it employs linear genomes; it performs an ontogenetic mapping from genotype to phenotype; and it uses a grammar to dictate legal structures in the phenotypic space.

Regarding the use of grammars, O'Neill and Ryan [9] state that they provide a simple, yet powerful, mechanism that can be used for the description of any complex structure such as languages, graphs, neural networks, mathematical expressions, molecules compounds. This was the main reason why Grammatical Evolution was chosen as the main and default Evolutionary Algorithm of the jGE Library.

Instead of trying to evolve computer programs directly, which is the case in Genetic Programming [3], [4], Grammatical Evolution uses a variable length linear "genome" which governs how a Backus Naur Form grammar definition is mapped to an executable computer program.

Grammatical Evolution takes the approach that a Genotype must be mapped to a Phenotype, like some other former approaches (Genetic Algorithm for Developing Software [11]), but it does not use a one-to-one mapping, and moreover it evolves individuals that contain no introns. It uses a Genetic Algorithm to control what production rules are fired when there are more than one choice for a Backus Naur Form non-terminal symbol [12].

In natural biology, there is no direct mapping between the genetic code and its physical expression. Instead, genes guide the creation of proteins which affect the physical traits either independently or in conjunction with other proteins [12]. Grammatical Evolution treats each transition as a "protein" which cannot generate a physical trait on its own. Instead, each one protein can result in a different physical trait depending on its position in the genotype and consequently, the previous proteins that have been generated.

Grammatical Evolution uses all the standard operators of Genetic Algorithms, plus two new operators: Prune and Duplicate. The gene duplication is analogous to the production of more copies of a gene or genes, in order to increase the presence of a protein or proteins in the cell of a biological organism. The gene pruning reduces the number of introns in the genotype and according to [12] it results in dramatically faster and better crossovers (Later research questions the usefulness of this operator because of the important role of introns [10]).

The main advantages of Grammatical Evolution according to Ryan [12] are the following:

- It can evolve programs in any language;
- Theoretically, it can generate arbitrary complex functions; and
- it has closer biological analogies to nature than Genetic Programming.

But Grammatical Evolution is, like Genetic Programming, subject to problems of dependencies [12]. For example, the further a gene is from the root of the genome, the more likely it will be affected by the previous genes. Ryan *et al*. [12] suggest the biasing of individuals to a shorter length and the progressive generation of longer genomes.

A well known and freely available implementation of Grammatical Evolution in the C++ language is libGE from Nicolau [6]. Characteristics of libGE are presented in [6] and [9].

2 jGE - Architecture

2.1 libGE vs. jGE

The libGE library is an implementation of the Grammatical Evolution system written in the C++ language. A recent version is the 0.26 beta 1, 3 March 2006 [6]. libGE implements the Grammatical Evolution mapping process. It can be used by an evolutionary computation algorithm in order to map the genotype (the result of the search algorithm) to the phenotype (the program to be evaluated). As Nicolau [6] says in the documentation "On its default implementation, it maps a string provided by a variable-length genetic algorithm onto a syntactically-correct program, whose language is specified by a BNF (Backus-Naur Form) context-free grammar."

Our implementation of the Grammatical Evolution system, the jGE Library, uses the Java programming language.[1] The main difference between jGE and libGE is that jGE incorporates the functionality of libGE as a component and provides implementation of the Search Engine as well as the Evaluator. Namely, as will be shown below, the jGE is a more general framework for the execution of Evolutionary Algorithms. Indeed, it still provides, like libGE, the feature of using any other Search Engine and Evaluator beyond that already provided by default in jGE. Individual components of the jGE, such as the GE Mapping Mechanism, the BNF Parser, and the Mathematical Functions classes, may also be used separately for special purpose projects.

Another main difference between jGE and libGE is the goal of each project. libGE provides an implementation of the Grammatical Evolution mapping process. On the other hand, the goal of the jGE Project is the development of a general Evolutionary Algorithms Framework which facilitates the incorporation and evaluation of Evolutionary techniques; and the incorporation of agent-oriented principles to develop implementations for parallel distributed systems.

2.2 Overview of jGE (v0.1)

The main idea behind the development of the jGE Library (current version is 0.1; it can be downloaded at [16]) is to create a framework for evolutionary algorithms which can be extended to any specific implementation such as Genetic Algorithms, Genetic Programming and Grammatical Evolution. This means that instead of using a mapper-centric approach like libGE, jGE uses a GA-oriented approach. Namely, the libGE instead of being just the implementation of the mapping mechanisms between the Search Engine and the Evaluation Engine, it provides libraries for both of these components. This means that someone using jGE is able to specify the core strategy of the evolutionary process by selecting the following parameters:

- the desired implementations of the genetic operators (selection, crossover, mutation, etc.);
- the genotype to phenotype mapping mechanism;
- the evaluation mechanism;
- the initial population;
- the initial environment (although currently not yet implemented).

The purpose of the last parameter is to allow the developer to specify an environment in which the

[1] Ghanea-Hercock [1] lists several advantages of using Java for the development of Evolutionary Algorithms applications: automatic memory management, pure object-oriented design, high-level data constructs (e.g. dynamically resizable arrays); platform independent code; and the availability of several complete EA libraries for EA systems.

population is living, and to influence not only the creation of new generations but the phenotype of the individuals as well, their growth process, and finally their own genotypes before they reproduce new offspring.

The problem specification and evolutionary strategy can be created in an external XML file which is loaded by the core mechanism of the jGE framework. The core mechanism is then responsible for allocating and executing the appropriate actions and directives, and to produce the final results.

2.3 Components of jGE

The following diagram (Figure 1) shows the main components of the jGE Library.

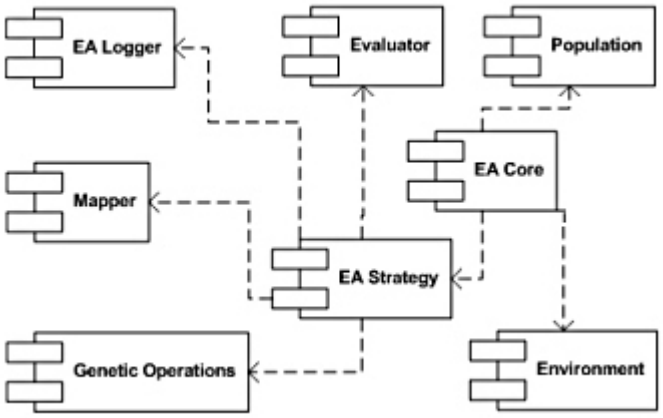

Figure 1: Component Diagram of the EA framework.

EA Core This is the main entry to the platform which loads the necessary files and classes. It loads the EA strategy to be executed and is responsible for checking the supported features and configuration possibilities of the EA strategy.

EA Strategy This supports the problem and evolutionary process specification. It is responsible for selecting the appropriate implementations of each one of the configurable elements: Genetic Operations, Mapper, Evaluator, and Logger.

Population This is a collection of classes which abstracts a real world population.

Environment This is a collection of classes which abstracts a real world environment.

Genetic Operations This component contains different implementations of the genetic operations such as selection, crossover, mutation etc.

Mapper This is responsible for the mapping of genotype to phenotype. Any specific mapping process can be implemented and executed by the EA strategy.

Evaluator The evaluator is used by the EA Strategy to assign a fitness value to the phenotype of an individual.

EA Logger This keeps track of the evolutionary process and stores the data for monitoring of the evolution of the individuals and for later use; that is, the creation of statistical results.

3 jGE - Design and Implementation

3.1 Description of the jGE v0.1 Packages

This section briefly describes the packages of the jGE v0.1 and further details of one of its components, the Genetic Operations package. Even though jGE is focused on the implementation of the Grammatical Evolution system, it contains all the necessary functionality for the execution and construction of other Evolutionary Algorithms as well. Currently, as well as GE, two other EC algorithms are implemented: Standard GA and Steady-State GA.

jGE decomposes and implements some services which are required by EC algorithms and provides functionalities for the ad-hoc implementations of other evolutionary based systems. In version v0.1, the library is concentrated on Grammatical Evolution. But this library can also be used by any other Java System for the creation of evolutionary algorithms as well as for other functionalities such as the parsing and representation of BNF Grammar definitions, the compilation and execution of Java programs, and the generation of random numbers in specific ranges.

The classes of the library are organised in the following packages. The detailed description of each class and its services can be found in the Java Documentation of the library (jGE v0.1 Documentation, see [16]):

Package: bangor.aiia.jge.core

This contains the core classes of the jGE library. These classes implement the Grammatical Evolution algorithm and some proof of concept experiments. Indeed, the interfaces of this package define the required functionality which must be provided by the evaluation and mapping components of the system.

Package: bangor.aiia.jge.population

This package contains the classes which represent the population of an evolutionary algorithm. A Population is a collection of Individuals and each Individual has a Genotype and a Phenotype.

Package: bangor.aiia.jge.evolution

The package *evolution* contains implementations of GA algorithms and classes which decompose the main operations of such algorithms (e.g. Crossover, Mutation, etc.). These operations are implemented as static methods and each class provides a collection of different variations. The classes include: crossover; duplication; genesis; mutation; pruning; selection. For example, the class Crossover currently provides a Standard One Point Crossover function but in forthcoming versions of the library, other crossover variations will be added.

Package: bangor.aiia.jge.ps

The problem specifications package contains implementations of problem specifications which can

be used for the evaluation of the individuals during an EA run. Each time a new problem is examined, a new problem specification class has to be created which assigns the Fitness Value to the Individuals of a Population. Classes are *HammingDistance* and *SymbolicRegression*.

Package: bangor.aiia.jge.bnf

This package contains all the necessary classes for the loading of a BNF Grammar definition and its representation as Java objects.

Package: bangor.aiia.jge.util

The package *bangor.aiia.jge.util* contains some helper classes. These classes implement common services used by other classes of the jGE library, like compilation and execution of java code, logging, stochastic functions, etc.

3.2 Speed Improvements on the Compilation/Execution of Java Code

The first version of the Evaluation component used the *JavaCompiler* class to evaluate the Java programs (phenotypes). This compiles (using the javac.exe compiler), and executes (using the java.exe runtime), once in each generation of a run, the dynamically created java source code which are the phenotypes of all the individuals of the population.

This is an extremely time consuming task and for problems such as Symbolic Regression (see section Experiments below), this is the most important factor which effects the execution speed. In each Symbolic Regression experiment the compilation/execution is taking place once when a new run starts (for the creation of the initial population) and once in each generation (during the evaluation of the individuals of the population).

Although the time complexity with respect to the compilation/execution of java code is linear ($\Theta(N)$, where N is the number of generations of a run), it is a significantly time consuming task which can significantly degrade overall performance. Moreover, other problems will have a higher rate of growth of execution time if they need to frequently use the source code compilation and bytecode execution tasks.

For the above reasons, some experiments were conducted into alternative methods regarding the compilation and execution of Java code. The experimental evidence (see [16]) leads to the conclusion that a much better solution than using javac.exe and java.exe is the following setup:

a) Use of the Jikes compiler [2] for the compilation of the java source code.

b) Utilization of the Dynamic Class Loading and Introspection features of the Java Virtual Machine (*ClassLoader* class, and the Reflection API).

Jikes is an open source Java compiler written in the C++ language and translates Java source files into the bytecode instructions set and binary format defined in the Java Virtual Machine Specification. Jikes has the following advantages as noted in the official web site of the compiler: open source; strictly Java compatible; high performance; dependency analysis; constructive Assistance.

The Java *ClassLoader* is an important component of the Java Virtual Machine which is responsible for finding and loading classes at runtime. It loads classes on demand into memory during the execution of a Java program. Furthermore, it is written in the Java Language and can be extended in order to load Java classes from every possible source (local or network file system, network resources, etc.). Using both the *ClassLoader* and the Reflection API, it is possible to perform the loading of Java bytecode and its execution from inside of any Java program using the same instance (process) of JVM.

In the current version (v0.1), the jGE Library provides the option of using either the Sun JVM or the IBM Jikes for the execution and compilation of Java code.

3.3 Genetic Operations Component

One of the most interesting and useful components of the jGE is the Genetic Operations component. Its classes implement various versions and types of the genetic operators as static methods. In this way, ad-hoc implementations of evolutionary algorithms can easily access the various genetic operations and use them in different combinations. Currently the following operators are implemented:

- **Genesis:** random creation of an initial pool of binary string genotypes; and random creation of an initial population of individuals.
- **Selection:** roulette wheel selection; rank selection; *N* best and *M* worst selection.
- **Crossover:** standard one-point crossover for fixed-length genotypes; standard one-point crossover for variable-length genotypes.
- **Mutation:** standard one-point mutation.
- **Duplication:** standard duplication.
- **Pruning:** standard pruning.

An abstract class *EvolutionaryAlgorithm* defines common properties and behaviours for evolutionary algorithms like Genetic Algorithms, Genetic Programming, and Grammatical Evolution. An Evolutionary Algorithm simulates the biological process of evolution. The evolution unit of this process is the population as Darwinism argues [5]. The basic strategy of an Evolutionary Algorithm is the following:

---

1. Initial Population Creation.
An initial Population is randomly created in case an already initialised population is not given to the algorithm.

2. In each Generation the following actions are executed:

- Competition (Evaluation of the Individuals of the Population).

- Selection (The individuals to mate).
- Variation (Crossover, Mutation, Duplication, Pruning, etc.).
- Reproduction (Creation of the new population, the offspring, which replaces the old population).

The subclasses of this class must implement the concrete steps of the above strategy in order to provide specific versions of Evolutionary Algorithms.

Further, two evolutionary algorithms have been implemented: the Standard Genetic Algorithm and a version of a Steady-State Genetic Algorithm. For the former, the following process is implemented:

1. Randomly Initialise the population, P (if not one given).
2. Perform Fitness Evaluation of the initial individuals in P.
3. Create an empty new population, P'.
4. Until P' is full:
- Select two individuals from P to mate using Roulette Wheel Selection.
- Produce two offspring using standard one-point crossover with probability P_c.
- Perform Point Mutation with probability Pm on the two offspring.
- Perform Duplication with probability P_d on the two offspring.
- Perform Pruning with probability P_p on the two offspring.
- Add the two offspring into P'.

5. Replace P with P'.
6. Perform Fitness Evaluation of individuals in P.
7. Repeat steps 3 until 6 until termination criteria are met (solution found or max generations exceeded).
8. Return the best individual (solution) in current population, P.

For the Steady-State Genetic Algorithm (SSGA), the main idea is that a portion of the population P survives in the new population P' and that only the worst individuals are replaced. Namely, a few good individuals will mate and their offspring will replace the worst individuals. The rest of the population will survive. The portion of the population P that will be replaced in P' is known as the Generation Gap and is a fraction in the interval (0,1). The default implementation of SSGA uses a fraction, $G = 2/n$ (where n the size of the population). Namely, two individuals will mate and their offspring will replace the two worst individuals. In general, the number of the individuals which will be replaced in each generation is $G * n$. The SSGA process implemented by this class is the following:

1. Randomly Initialise the population, P (if not one given).
2. Perform Fitness Evaluation of the initial individuals in P.
3. Create an empty population, P'.
4. Until new offspring = $G * n$
- Select two individuals from P to mate using Roulette Wheel Selection.
- Produce two offspring using standard one-point crossover with probability P_c.
- Perform Point Mutation with probability P_m on the two offspring.
- Perform Duplication with probability P_d on the two offspring.
- Perform Pruning with probability P_p on the two offspring.
- Add the two offspring into P'.

5. Add the best n - ($G * n$) individuals of P into the offspring P'.
6. Replace P with P'.
7. Perform Fitness Evaluation of the individuals in P.
8. Repeat steps 3 until 7 until termination criteria are met (solution found or max generations exceeded).
9. Return the best individual (solution) in current population, P.

In the case where $G * n$ is not an even integer, then the larger even integer less than $G * n$ and larger than 0 will be used.

The class *GrammaticalEvolution* implements the default version of the Grammatical Evolution (with a minor exception regarding the steady state replacement mechanism as mentioned below). The default implementation as described by O'Neill and Ryan, uses a Steady-State replacement mechanism such that two parents produce two children, the best of which replace the worst individual in the population only if the child has a greater fitness than the individual to be replaced. Our implementation uses a slightly different replacement mechanism which is described above in the SSGA process. Also, there is the option to use a Generational replacement mechanism like in Standard GA.

Regarding the configuration of a Grammatical Evolution run, O'Neill and Ryan suggest the following: a typical wrapping threshold is 10; the size of the codon is 8-bits; and typical probabilities are: crossover – 0.9; mutation – 0.01; duplication – 0.01; pruning – 0.01.

The above configuration is the default of the *GrammaticalEvolution* class. Further, this implementation uses the following default values: max. generations: 10; searching mechanism: Steady-State GA; Generational Gap of the Steady-State GA, $G = 2/n$ (n = the population size).

The next section describes some proof-of-concept experiments performed with the jGE Library and presents the results.

4 Experiments

Experiments in [13] and [14] show that Grammatical Evolution is able to solve Symbolic Regression, Trigonometric Identity, and Symbolic Integration

Problems. The adoption of the Steady State approach [14] dramatically improves the performance of the Grammatical Evolution algorithm, making it as efficient in the mentioned problems types as the Genetic Programming algorithm. Also, [7] demonstrated the generation of multi-line code in the classical Santa Fe Ant Trail problem. Indeed, the last experiment showed that Grammatical Evolution outperforms Genetic Programming ([3] and [4]) in this specific problem when GP does not use the solution length fitness measure [7].

The following subsections briefly mention the types of proof-of-concept experiments which have been conducted with jGE and which lead to the results and findings which are discussed in the last section. Detailed set-ups and results of these experiments are available at [16].

4.1 Hamming Distance Experiments

The Hamming Distance problem involves the finding of a given binary string. The target string was: 11100011100010101010101010101010. For this problem, Grammatical Evolution, Standard GA, and Steady-State GA were compared.

4.2 Symbolic Regression Experiments

Symbolic Regression problems are problems of finding some mathematical expression in symbolic form that matches a given set of input and output pairs. The particular function examined was the following: $f(x) = x^4 + x^3 + x^2 + x$.

4.3 Trigonometric Identity Experiments

The particular function examined was cos 2*x*, and the desired trigonometric identity were $1\text{-}2\sin^2 x$ (for this reason the unary operator Math.cos was not included in the BNF Grammar of this problem).

The objective of these experiments was to find a mathematical expression identical to the examined function.

5 Discussion

The results of the above experiments with jGE confirmed three expected findings. First, that jGE using Grammatical Evolution is able to produce useful solutions even though these are not the best possible or the fastest. Secondly, that different set-ups and configurations of the searching and evaluation mechanisms have a significant impact on the quality and speed of the solution. Finally, the need for more than a standard PC's processing power is prominent.

The above confirmed expectations show that further research and work in this project toward its mentioned goal and objectives (see section on motivation) is promising. The above findings will guide the next steps in this project. Namely, the next version of jGE will provide implementations of more Genetic Operators which will facilitate experiments in a larger range of possible configurations. The need for more processing power will be tackled with an agent-based framework capable of executing transparently on many machines (i.e. a parallel distributed system). Finally, this agent-based framework will guide the later development of an agent-oriented evolutionary system.

References

[1] GHANEA-HERCOCK, R. (2003) *Applied Evolutionary Algorithms in Java*. New York, NY: Springer.

[2] IBM Corporation (2004), *Jikes 1.22*. United States: NY. Available from: http://jikes.sourceforge.net.

[3] KOZA, J.R. (1992) *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. Cambridge, MA: MIT Press.

[4] KOZA, J.R. (1994) *Genetic Programming II: Automatic Discovery of Reusable Programs*. Camb., MA: MIT Press.

[5] MAYR, E. (2002) *What Evolution Is*. London: Phoenix.

[6] NICOLAU, M. (2006), *libGE: Grammatical Evolution Library for version 0.26beta1, 3 March 2006*. Available from: http://waldo.csisdmz.ul.ie/libGE/libGE.pdf.

[7] O'NEILL, M. and RYAN, C. (1999) Evolving Multi-line Compilable C Programs. In *Proc. of the Second European Workshop on Genetic Programming, 1999*, pp. 83-92.

[8] O'NEILL, M. and RYAN, C. (2001) Grammatical Evolution. *IEEE Transactions on Evolutionary Computation* 5(4), 349-358.

[9] O'NEILL, M. and RYAN, C. (2003) *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. USA: Kluwer Academic Publishers.

[10] O'NEILL, M., RYAN, C. and NICOLAU, M. (2001) Grammar Defined Introns: An Investigation into Grammars, Introns, and Bias in Grammatical Evolution. In *Proceedings of GECCO 2001*.

[11] PATERSON, N. and LIVESEY, M (1997) Evolving caching algorithms in C by GP. In *Genetic Programming 1997*, pages 262-267. MIT Press.

[12] RYAN, C., COLLINS, J. J. and O'NEILL, M. (1998) Grammatical Evolution: Evolving Programs for an Arbitrary Language. Lecture Notes in Comp. Sci. 1391. *First European Workshop on Genetic Programming 1998*.

[13] RYAN, C., O'NEILL, M. and COLLINS, J. J. (1998) Grammatical Evolution: Solving Trigonometric Identities. In *Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets held in Brno, Czech Republic June 24-26 1998*, pp. 111-119.

[14] RYAN, C. and O'NEILL, M. (1998) Grammatical Evolution: A Steady State Approach. In *Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms, 1998*, pp. 419-423.

[15] TEAHAN, W.J., AL-DMOUR, N., TUFF, P.G. (2005) On thought, knowledge, evolution and search. In *Proceedings of Computer Methods and Systems CMS'05 Conference held in Krakow, Poland 14-16 November 2005*.

[16] UWB School of Informatics (2006), *Java GE (jGE) Official Web Site*. United Kingdom: Bangor. Available from http://www.informatics.bangor.ac.uk/~loukas/jge.