Exploring the Possibilities and Restrictions of Genetic Programming in Java Bytecode

Stefan Klahold Steffen Frank
Department of Computer Science
University of Dortmund
D-44221 Dortmund, Germany
klahol00@marvin.informatik.uni-dortmund.de
steffen.frank@ruhr-uni-bochum.de

Robert E. Keller Wolfgang Banzhaf
Systems Analysis
Department of Computer Science
University of Dortmund
D-44221 Dortmund, Germany
keller@icd.de
banzhaf@icd.de

ABSTRACT

The Java bytecode (Meyer 1997)(Lindholm and Yellin 1997) has various interesting features that differ from other programming languages. Thus, genetic programming (GP) with bytecode individuals is investigated in this contribution. The main issue of this paper is to reveal where the bytecode represents possibilities and restrictions in the context of GP. To that end, the GP system JAPHET (JAVA bytecode program optimization with help of evolutionary techniques) (Frank and Klahold 1998) has been created and used for experiments presented here.

1 Introduction

So far, GP in Java bytecode – called "bytecode" hereafter – has been investigated independently by (Lukschandl *et al.* 1998) and (Frank and Klahold 1998). This topic is interesting for several reasons, the major ones being given subsequently.

- First of all, the bytecode is efficiently platform-independent since it runs on any Java Virtual Machine (VM) (Meyer 1997)(Lindholm and Yellin 1997) without the need for time-consuming recompiling, and there are VMs available for many different platforms.
- The bytecode is object-oriented (Budd 1991) and can be used to evolve complexly structured programs. The evolution of object-oriented structures is discussed in (Bruce 1996).
- Bytecode programs are normally generated by a compiler, and such compilers exist for many languages including – of course – Java, and also Scheme, Eiffel, Ada and others. Thus, the language used for programming a problem solution which will ultimately be bytecode-represented can be one of those mentioned above.
- Due to its proximity to assembler-like languages, the bytecode can be efficiently processed.
- The bytecode is defined in the Java virtual machine specification (Lindholm and Yellin 1997), and the standardization process is on its way.

Subsequently, the genotypic representation, as derived from the bytecode properties, will be presented. Then, empiric investigations will be described.

2 Java bytecode and the virtual machine

Usually, a **bytecode program** is generated from a Java program by a compiler as shown in fig. 1.

The resulting bytecode program is stored in **class files**. All class files are then transformed into a **VM-internal representation (VMIR)** that is processed by the VM. During processing and due to it, the VM permanently changes its internal state. The VM specification only defines how the single bytecode operations change the internal state of the VM. It does not define how the state change is implemented.

There are different ways of executing the VMIR on the underlying platform. An interpreter, for instance, can iteratively take a certain VMIR segment, translate it into machine code, and then the CPU executes the machine code. Subsequently, the interpreter takes over again, and the cycle starts again. Another approach is to compile the VMIR completely into machine code and then to execute this code. Alternatively, a just-in-time (**JIT**) compiler can be used.

The class files of which a bytecode program consists all have the same specified structure. This structure is designed to be efficiently transmittable via a network. Thus, it is simple and compact.

A class file consists of different **sections**. Especially, there is a section called the **constant pool** which holds all the con-

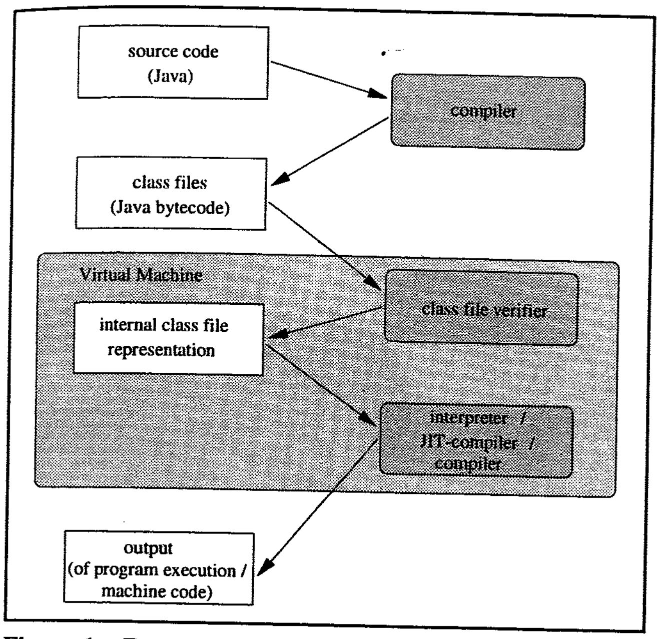

source code (Java)
compiler
class files (Java bytecode)
Virtual Machine
class file verifier
internal class file representation
interpreter / JIT-compiler / compiler
output (of program execution / machine code)

Figure 1 Processing of Java bytecode programs.

stants in the class, like constant strings, references to the names of other classes and names of variables. Another subsequently interesting section contains the representation of the *methods* (Budd 1991).

The VM's internal representation is *stack-based.* Operands for instructions and results of instructions are pushed on a stack.

3 GP with JAPHET

JAPHET directly evolves Java bytecode program individuals. Thus, the above-described structure of a bytecode program is the individual structure as shown in fig. 2. Since bytecode is a linear structure, the genotypic representation is *linear* as, for instance, in (Keller and Banzhaf 1996) and (Nordin and Banzhaf 1995).

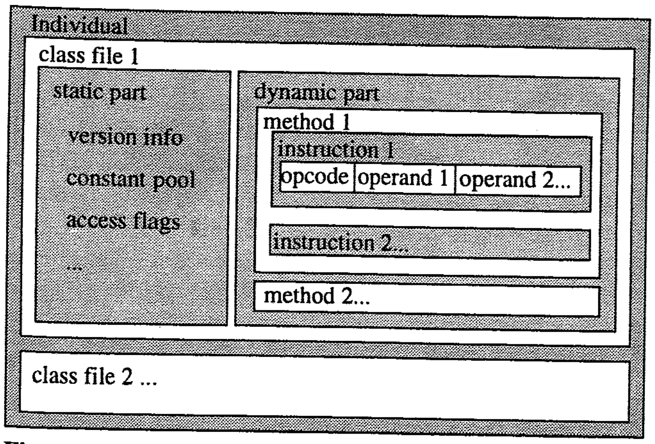


Figure 2 The individual structure used in JAPHET.

The above-displayed distinction between the **dynamic** and the **static** part of a class file is JAPHET terminology for how genetic operators treat the respective part. The static part is currently not affected by the operators. They only affect the dynamic part which is bytecode that represents methods. It is, however, possible to mark a method – which, by default, is dynamic – as static.

It is a non-trivial task to keep, for instance, the constant pool consistent when applying a variation operator – like mutation – to it, since many structural constraints apply to the pool. Thus, in the current JAPHET version, the dynamic/static distinction is used.

A method is an **instruction** sequence. An instruction itself is a symbol sequence beginning with an **opcode** at **symbol position** zero, followed by zero or more **operands** in a sequence beginning at symbol position one.

The *recombination operators* work on method level or on instruction level.

With respect to the **instruction level**, there are several two-point crossover and one-point crossover variants. A point corresponds to a symbol position. One variant randomly selects the crossover point(s). Another variant swaps method parts having the same **length**, that is number of instructions. A third variant swaps method parts beginning at the same crossover point, e.g., symbol position three.

On the **method level**, there is a recombination operator that randomly selects one or more methods to be swapped.

Note there is no recombination on the **opcode/operand level**. This has been established in order to reduce the amount of corrupted class files representing programs that produce errors at run-time. Thus, all individuals resulting from crossover operations are syntactically legal.

A *mutation operator* is also used. It modifies an opcode or an operand. The mutant is syntactically legal.

The used *selection operators* are fitness proportionate selection, rank selection and tournament selection.

With respect to *creation* of the initial generation, a purely random approach is prohibitive since the VM is stack-based. Thus, for instance, a purely randomly created method will probably produce a runtime error. It is a hard task to random-create **semantically legal** – that is, non-run-time-error producing – individuals. Therefore, another creation procedure is used and shown in figure 3.

A semantically legal class file *defined by the user* is taken. Then, short segments of bytecode are inserted. These segments have a special property: after their execution, the stack of the VM is in the same state as it was before their execution. Thus, all created individuals are legal.

When working with a GP system, there is a need for problem-specific fitness functions, genetic operators, selection operators and creation procedures. According interfaces in our adaptation of the basic GP algorithm by (Koza 1992) are shown in figure 4.

As the JAPHET system is implemented in Java, a user must write a Java class for a desired non-default interface.

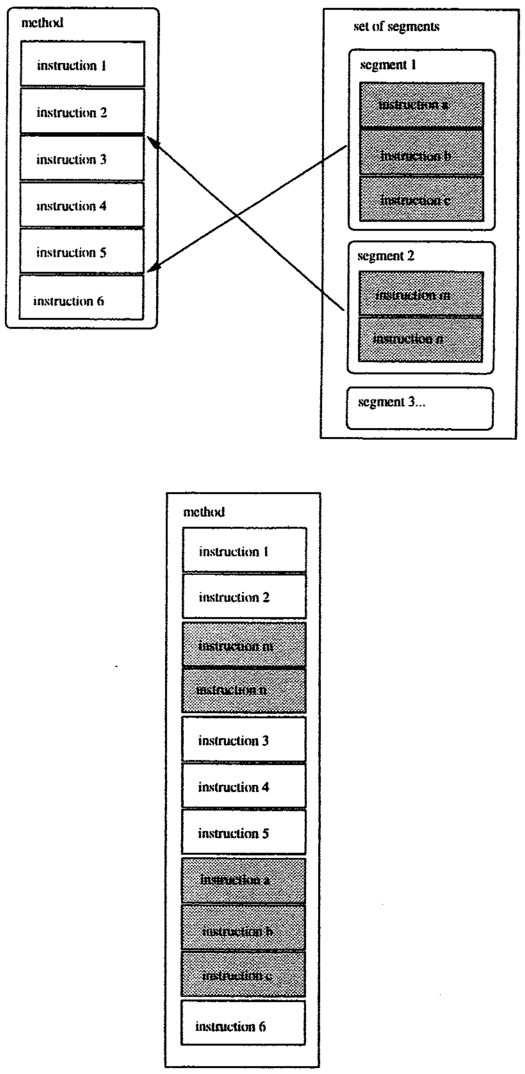


Figure 3 Building an individual for the initial generation.

4 Experiments

The first experiment shows that the GP algorithm works for bytecode, and that the evolved programs are platform-independent. The problems are function regressions. The functions are integer-coefficient polynomials of a degree less than four.

Quadratic polynomials are found fast by the system. For example, $f(x) = x^2$ was always found in less than 10 generations with 100 individuals as population size. A genome-length explosion characteristic for many GP systems takes place.

Also, the system evolves methods that use stack-manipulation sequences in order to create coefficient values. This, by human standards, unnatural way of programming is typical of GP systems.

Polynomials of degree three and four most often are approximated instead of found. The approximations have satisfying quality when using 500 generations with population

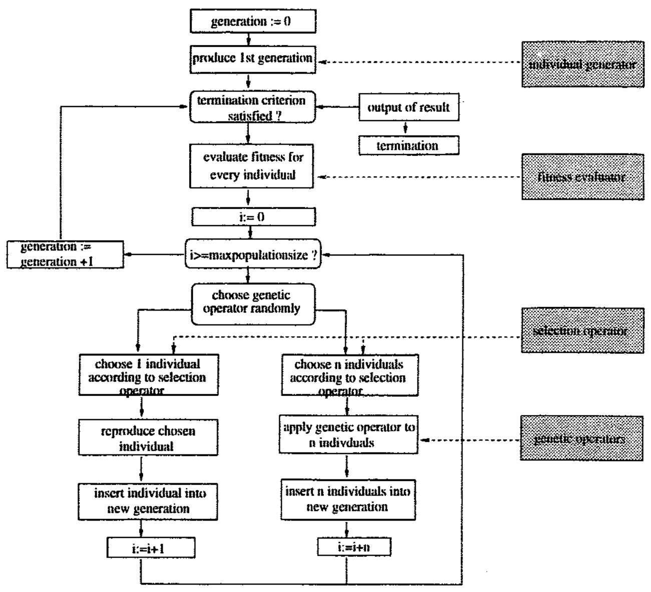

generation := 0
produce 1st generation
individual generator
termination criterion satisfied ?
output of result
termination
evaluate fitness for every individual
fitness evaluator
i:= 0
generation := generation +1
i>=maxpopulationsize ?
choose genetic operator randomly
selection operator
choose 1 individual according to selection operator
choose n individuals according to selection operator
reproduce chosen individual
apply genetic operator to n indivduals
genetic operators
insert individual into new generation
insert n individuals into new generation
i:=i+1
i:=i+n

Figure 4 Interfaces for the variable components of the basic GP algorithm.

size 100.

With respect to answering the question of platform independence, virtual machines of several different platforms were used to run the evolved programs. No run produced a runtime error, and all results were reproduced.

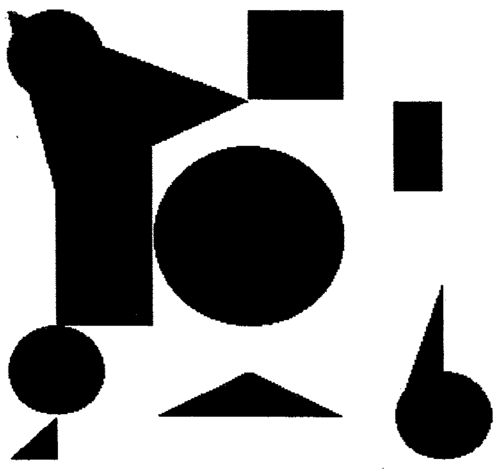


Figure 5 Original image.

The second experiment investigates the usage of external method calls in evolved programs. The problem is the approximation of a given image as shown in figure 5.

The system is granted access to a class library offering methods for drawing geometric shapes. These methods are not part of the evolution process. Just their calls are subject to evolution.

The best result is shown in figure 6.

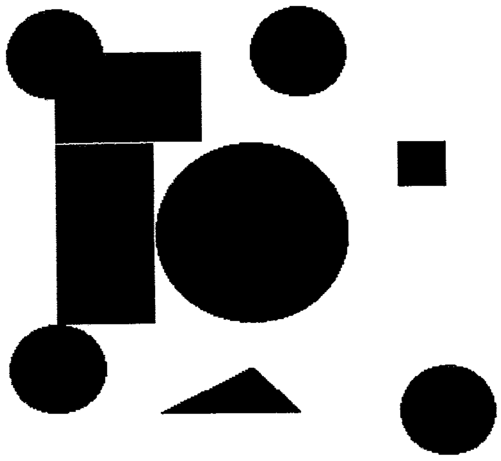


Figure 6 Best result for the third experiment.

It shows a good approximation of the original image. However, this result is not representative. A more representative result is shown in figure 7.

The image consists of circles only. The tendency to favor circles can be explained if one takes a look at the form of the method calls.

A circle is described in the used class library by its center and a radius, a rectangle by two opposite points, and a triangle is specified by all three points. As the points in the library are represented as (x, y) coordinates, there are three parameters to produce a circle, four for a rectangle and six for a triangle.

The statistics about the use of the method calls in the individuals show that the number of calls taking place depends on the number of parameters for that method. In many runs, the triangles vanish very early. The rectangles tend to stay longer but they are eliminated also in many cases. These shapes only survive when they are able to add a positive effect on the fitness within the first generations of the evolution process.

This phenomenon is due to the structural complexity of a method call. As all parameters for method calls are delivered via the stack of the VM, there is a better chance for the evolution of a valid stack when the number of parameters is low. When a method call has vanished from all the individuals it is nearly impossible to recreate such a call by a simple mutation operator because of the high structural constraints imposed on the stack.

The last experiment focuses on the possibility to user-define the structure of the result by creating a structured class as a starting point for the initial generation. The problem is

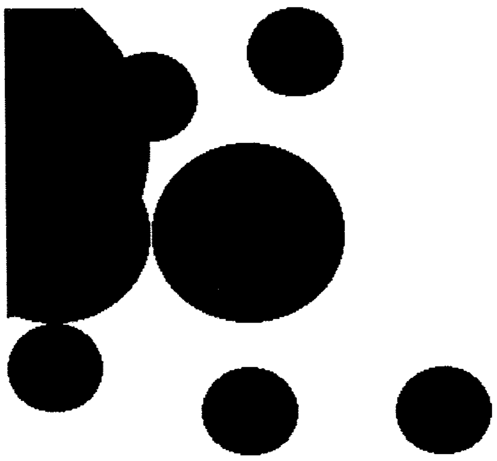


Figure 7 A more representative result for the third experiment.

simulating an LED with one digit divided into seven segments as shown in figure 8.

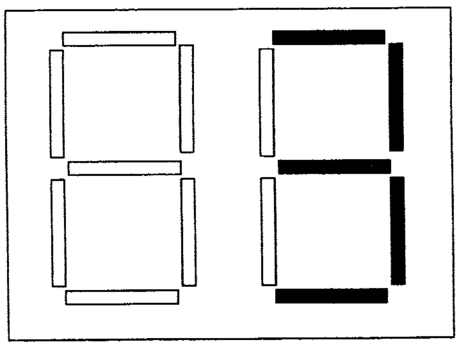


Figure 8 An LED with seven segments empty and showing the digit 3.

The aim is to find, for a given one-digit number, the corresponding LED representation. As setup, for each of the LED segments, one method is determined. There is one method which is not evolved that takes the seven results of the methods for the segments and produces the total result. Each of the seven subproblems is the same if the fundamental structure is regarded: produce an output (LED segment on or off) according to the input number.

A perfect result is found in less than 300 generations on average when using a crossover operator on method level that randomly selects one method to be swapped in each parent. This is opposed to selecting mutually corresponding methods, e.g., the two methods to be swapped both control LED segment number four. When using this operator, the average number of generations needed to find a perfect solution is about 400.

Most of the convergence-enhancing effect takes place in the early stages of the evolution. The potential explanation is that the evolved methods are not very specialized in the early stage. Thus, for instance, method six may also fit for subproblem 2. This is what gets exploited by the randomly selecting crossover operator.

The experiment also shows that method-level recombination makes sense when a problem is composed of identical subproblems. Method-level recombination reduces the average number of generations needed for finding a perfect solution from around 300 – needed when using instruction-level recombination – to around 280 with population size 100.

5 Conclusion

This contribution deals with possibilities and restrictions of GP in bytecode. It is possible to evolve platform-independent programs in Java bytecode. However, a restriction to this is the class file verification process (fig. 1). This process is not as entirely specified as the VM internal processing of the bytecode instructions. It is almost completely left to the implementation of a particular VM which kind of class file verification is established. Thus, an evolved bytecode program may not pass a specific VM's class file verifier.

Evolving programs with class file library method calls is possible. However, the number of method parameters is critical: more parameters mean a higher probability of generating an inconsistent VM stack.

It is possible and meaningful for certain problems to restrict the search space by predetermining the solution structure. However, this restriction must be carefully performed such that potentially good solutions lie within the restricted area.

6 Related and future work

A more detailed investigation into this contribution's topics can be found in (Frank and Klahold 1998). Future work must go into the elimination of the dynamic/static distinction. Furthermore, for prominent VMs, it must be assured that the class file verification process accepts the evolved individuals.

References

Bruce, W.S. (1996). Automatic generation of object-oriented programs using genetic programming. In: *Genetic Programming 1996: Proc. of the 1st annual Conf.* (J.R. Koza, D.E. Goldberg, D.B Fogel and R.L. Riolo, Eds.). MIT Press, Cambridge, MA. pp. 267–272.

Budd, Tim (1991). *An Introduction to Object Oriented Programming*. Addison-Wesley.

Frank, S. and S. Klahold (1998). *JAPHET - Ein System zur Untersuchung der Moeglichkeiten und Beschraenkungen fuer Genetisches Programmieren in Java Bytecode.* University of Dortmund, Fachbereich Informatik, Internal Reports.

Keller, R.E. and W. Banzhaf (1996). Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In: *Genetic Programming 1996: Proc. of the 1st annual Conf.* (J.R. Koza, D.E. Goldberg, D.B Fogel and R.L. Riolo, Eds.). MIT Press, Cambridge, MA. pp. 116–122.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA.

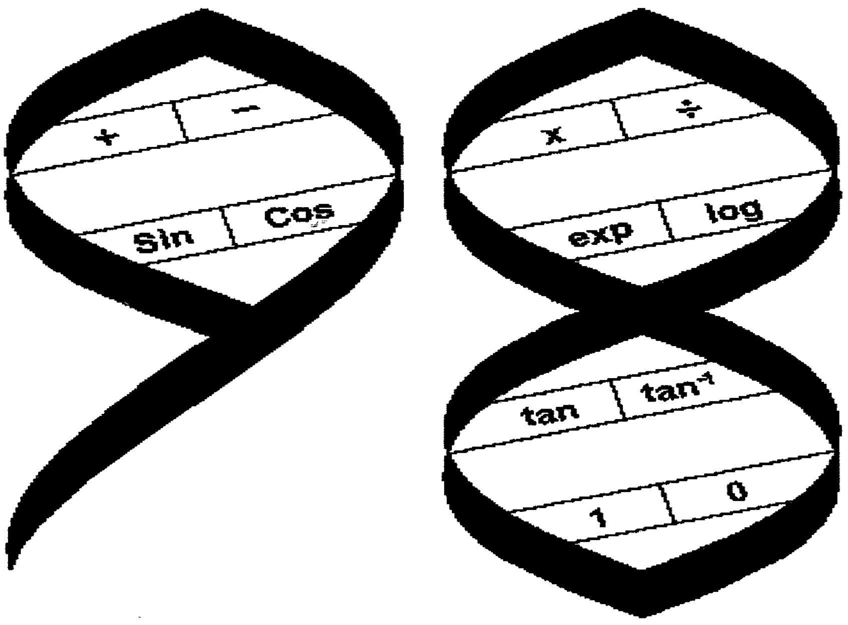
Lindholm, T. and F. Yellin (1997). *The Java Virtual Machine Specification.* Addison-Wesley.

Lukschandl, Eduard, Mangus Holmlund and Eirk Moden (1998). Automatic evolution of java bytecode: First experience with the java virtual machine. In: *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming* (Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty and Wolfgang Banzhaf, Eds.). The University of Birmingham, UK. Paris, France. pp. 14–16.

Meyer, J., Downing T. (1997). *The Java Virtual Machine.* O'Reilly and Associates, Inc., Sebastopol, CA.

Nordin, P. and W. Banzhaf (1995). Evolving turing-complete programs for a register machine with self-modifying code. In: *Genetic Algorithms: Proc. of the 6th Int. Conf.* (L. Eshelman, Ed.). Morgan Kaufmann, San Francisco, CA. pp. 310–317.

Late Breaking Papers at the Genetic Programming 1998 Conference University of Wisconsin – Madison July 22 - 25, 1998



Edited by
John R. Koza
Stanford University