DOI: http://dx.doi.org/10.7551/978-0-262-32621-6-ch133

CodeRouge: a Project to Evolve Life-like Autonomous Programs

Nawwaf N. Kharma[1] and William R. Buckley[2]

[1]ECE Dept., Concordia University, 1455 de Maisonneuve Blvd. O., Montreal, QC, Canada H3G1M8
[2]California Evolution Institute, San Francisco, CA, USA 94134
[1]kharma@ece.concordia.ca and [2]wrb@calevinst.org

Abstract

The aim of this project is to create a computational environment that allows for the design/evolution of programs with life-like behavior. By life-like behavior we mean programs whose main aim is to exist and reproduce within their environment, and exhibit other essential signs of life: homeostasis & adaptation, growth & open-ended evolution. In order to give digital organisms a functionality of use to humans, a program will also be able to carry out, and autonomously improve upon, a human defined activity. For many reasons, we have chosen to build a computational environment (in emulation) and a new redcode-like language, μRouge, which runs in it. This paper describes the concepts and instructions of this new language and provides an example highlighting some of its unusual characteristics.

Motivation

The existing redcode of Core War (Dewdney, 1984; Jones and Dewdney, 1984) allows for the development of novel algorithms through the direct manipulation of program instructions but, it does not allow for those novel algorithms to be shared among a population of programs. The current effort extends redcode to provide explicit facilities for such algorithm sharing, and so increasing evolutionary pressure on shared algorithms. Further, evolution in Core War redcode programs is ungoverned save solely the purpose of execution longevity. We propose an added level of selection within the model, by means of an auxiliary task; satisfaction of this task concomitant with steadily improving life scores is critical to the continued computation of the program. Ultimately, we wish to build a programming language and platform that allow for the evolution of programs with life-like characteristics- characteristics that are discussed below.

Existence

A program exists if its body is present in, or can deploy to, working memory (~ space), where it can access processing time (~ energy).

Metabolism

A program has a metabolism if it is using processing time to execute instructions. The environment is any space with a state that can be altered, computationally, which includes working memory as well as long-term memory.

Growth

Growth is an increase in the size of a program. The size of a program is the amount of working memory it occupies, when active (i.e., has a metabolism).

Reproduction

Reproduction of a program is the synthesis of a copy of the program identical/similar to all/part of the parent, a copy that occupies memory space outside the boundaries of the memory space of the parent. Copying occurs imperfectly.

Darwinian Reproduction. In this mode of reproduction, a program copies a compressed self-description of its *original* body to a free location in memory, also copies a compression/decompression routine (or *comdec*) to that location, then applies it to the compressed self-description, to complete the synthesis of the body of the child.

Lamarckian Reproduction. In this mode of reproduction, a program first creates a compressed self-description of its *current* body, then it copies that together with the comdec to a free location in memory. Hence, the program applies the comdec to the compressed self-description in order complete the synthesis of the body of the child.

Homeostasis

Homeostasis of a program is its ability to maintain or regain its functionality (possibly imperfectly) in response to a perturbation to its body or environment (e.g., input). To achieve this, we assert that (a) Minimal changes (or *mutations*) to either input data or instructions will have no impact on the meaning of that data or instructions; (b) greater (non-minimal) mutations to an instruction or datum can only result in proportional changes in the meaning of that instruction or datum; (c) no amount of mutation will render an instruction or datum meaningless; (d) instructions can be interpreted as data and data as instructions.

Adaptation

A program adapts by changing its body or that of its descendants in response to changes it senses in its body or environment. A program can spring-off mutated copies of itself, but it can also modify its own body during its existence.

Evolution

Both flavors of reproduction result in a diverse population-selection is needed. Selection can be explicit (and purposeful: see Utility) or implicit. Implicit selection occurs if, for example, smaller programs are allowed to replicate faster than larger programs. It is important that we do not unconsciously implicitly favor one kind of program over another. Also, to allow for open-ended evolution, our model permits the run-time definition of new instructions.

Utility

A particular part of a program (called *human load*) is assessed for its effectiveness in fulfilling a human-specified functionality, and for efficiency, represented by size.

Program Structure & Instruction Set

A program is made of a circular linked list of *program lines*. A program line has the format: <descriptor> <instruction-modifier><data source & destination><next instruction pointer><error detection code>. An *instruction* is a traditional instruction (INS), a new instruction (NIN), a data line (DTA) or a free line (FRE). An INS indicates a pre-defined instruction that transforms data (e.g., ADD). A NIN says that this instruction was defined by the program itself at run-time. DTA indicates data. FRE says that this space is free to be used as local working memory.

Instruction Set

Arithmetic & Logical Operations. *Arithmetic* addition (ADD), subtraction (SUB), multiplication (MUL) and division (DIV) are all implemented. *Logical* AND, OR (ORR), NOT and XOR are implemented. *Shift* left (SHL) and shift right (SHR) are also implemented.

Program Flow Control. An *unconditional jump* (JMP) and *conditional jumps* on zero (JMZ) and not-zero (JMN) are implemented. Also, jumps conditional on the equality (CMP) or inequality (SLT) of the first two operands are implemented. Finally, we implemented a jump (DJN) that first decrements the second operand then jumps to a location specified by the first operand only if the result of the decrement is not zero.

Self-Sensing Instructions. A critical subset of instructions are those providing a program with information about itself. These include: *age* (AGE), which returns the number of cycles that passed since the program was first instantiated; *size* (SIZ), which returns the number of lines a program is made of; *metabolism (*MET*)*, which is a measure of program activity- a function of the number of executed instructions; *homeostasis* (HOM), which is a measure of program health: freedom from errors; *growth* (GRW), which is a ratio of the program's current size to its initial size; offspring (OFS), which is the number of offspring a program created since its instantiation; number of *new instructions* (NNI), which is the number of new instructions (automatically defined functions, in fact) that were defined by the program since its instantiation; *fitness* (FIT), which is a measure of the performance of the human load.

Self-Modification Instructions. Another critical set of instructions are those that allow a program to alter its own code. *Edit* (EDT) allows for the modification of a line (or part thereof), at a given location within the program, in a particular manner. *Generate random* (GRN) inserts a randomly generated instruction or data line at a given location within the program. The *copy sequence* (COS) instruction and *move sequence* (MOS) instruction, respectively, copies *or* moves a number of contiguous program lines to a certain location within the program. Copying can occur w/without mutation, w/without compression/decompression, and by overwriting or down-shifting the lines at the insertion point.

Program Reproduction. There are two instructions that implement the two modes of replication described above: REP, which realizes *Darwinian* reproduction and CON, which implements *Lamarckian* reproduction.

New Instruction Definition. This allows for the run-time definition of new instructions, in an analogous fashion to automatic function definition (Koza, 1992).

Miscellaneous Instructions. A few other instructions (5) are added for pragmatic considerations.

Program Example

```
     MET       $F,      $G,      #0     ;metabolism
     OFS       $C,      #0,      #0     ;offspring
     NI1       $C,      $F,      $S     ;new inst. 1
     CMP       $C,      $S,      #0     ;ready?
     JMP       $O,      #0,      <T     ;if not, exit
     COS       $SR,     #LN,     $DS    ;copy subrtn
     EDT-REP   $ALT,    $DS+n,   #7     ;edit opr c
     SPL       $DS+e,   <SCT,    #0     ;new thread
O:   ADD       #1,      $PCT,    #0     ;continue
```

The example shows the use of program health measures, metabolism and offspring, in controlling both the selection of other subprograms for replication, the spawning of a new thread of execution corresponding to such a subprogram replication event, and a post-replication change to the compression-decompression algorithm used for subsequent subprogram replication tasks.

Here, the health measures metabolism (MET) and offspring (OFS) are combined by a new instruction (NI1) to generate a testable condition, stored in location $S. This condition then governs the spawning of a new process; if ready, a copy of the source code is placed at the destination address, and a targeted REP instruction of the copy is also modified, replacing the comdec. It is this modified program that is spawned by the SPL instruction.

References

Koza, J. R. (1992). Hierarchical automatic function definition in genetic programming. In Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems.

Dewdney, A. K. (May 1984). Computer Recreations. Scientific American.

Jones, D. G., Dewdney, A. K. (March 1984). Core War Guidelines. http://corewar.co.uk/cwg.txt.