TR - H - 126

An Evolutionary Approach to Program Transformation and Synthesis

Sung-Bae Cho
Thomas S. Ray

1994. 2. 7

ATR人間情報通信研究所

〒619-02　京都府相楽郡精華町光台2-2　☎ 0774-95-1011

ATR Human Information Processing Research Laboratories
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan
Telephone: +81-774-95-1011
Facsimile: +81-774-95-1008

An Evolutionary Approach to Program Transformation and Synthesis[*]

Sung-Bae Cho and Thomas S. Ray
ATR Human Information Processing Research Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

January 31, 1995

Abstract

Efficiency is a problem in automatic programming—both in the programs produced and in the synthesis process itself. This paper presents a framework for using evolutionary mechanisms to guide program synthesis. A particular implementation of the framework, called Tierra, is described. Given a naive program and some limits on system resources, Tierra generates mutated programs and evolution proceeds by natural selection as the programs compete for CPU time and memory space. By applying the evolutionary mechanisms, Tierra has guided the automatic implementation of an efficient self-replicating program. The system is under continuing development, it is viewed more as a research tool than a prototype programming assistant, however, the performance of the system at present gives some hope for the ultimate feasibility of such systems.

1 Introduction

Since the early days of computing, effort has been put into automating more and more of the programming process. The ultimate objective of automatic programming would allow a user to simply say what is wanted and have a program produced completely automatically. However, fully automatic programming is too big a problem to be solved in a single step. In order to provide avenues of attack, researchers have cut the problem down in a number of different ways.

1.1 Automatic programming revisited

A variety of approaches have been taken in facing this problem, including extending language development to higher level languages [1, 2], many attempts involving the application of artificial intelligence techniques like deductive synthesis [3, 4], and using transformations to support program construction [5, 6]. This section reviews each of the three approaches briefly.

First, the archetype of a higher level language is SETL, which supports most of the standard constructs of any Algol-like programming language. In addition, it supports two

[*]This work was supported by grants CCR-9204339 and BIR-9300800 from the United States National Science Foundation, a grant from the Digital Equipment Corporation, and by the Santa Fe Institute, Thinking Machines Corp., IBM, and Hughes Aircraft.

convenient universal data structures: tuples and sets. For example, a mapping is treated as a set of 2-tuples. SETL also supports the use of universal and existential quantifiers in a program. For example, the following is the form for a SETL statement for performing some computation on every element of the set S.

$$(\forall x \in S)\ldots\textbf{end}\ \forall;$$

The goal of providing such expressive facilities is to free the programmer from having to think about the detailed design of data structures. This decrease in what the programmer has to worry about is a key to the productivity gains that should be obtained by the use of higher level languages.

Second, deductive program synthesis is based on the observation that constructive proofs are equivalent to programs because each step of a constructive proof can be interpreted as a step of computation. A constructive theorem prover can be used to derive a program from its specification as follows. Suppose that the program to be derived takes an input x and produces an output y. Further, suppose that the specifications of the program state that a precondition $\mathrm{P}(x)$ should be true of the input and that a postcondition $\mathrm{Q}(x, y)$ will be true of the output. This specification is converted into the following theorem, which is given to the theorem prover.

$$\forall x\ \exists y\ [\mathrm{P}(x) \Rightarrow \mathrm{Q}(x, y)]$$

The process of constructive proof, in effect, discovers a method for finding y for any given x.

Finally, a program transformation takes a part of a program and replaces it with a new transformed part. Programmers are encouraged to postpone questions of efficiency and first write their programs as clearly as possible. These programs are then transformed into efficient versions by applying transformations. Typically, a program transformation is correctness preserving, in that the new part of the program computes the same thing as the old part. The purpose of a transformation is to make the part better on some scale, e.g., more efficient or more concrete. For example, a transformation might be used to replace "`X**2`" with "`X*X`."

1.2 Evolutionary approach

One serious problem in automatic programming is efficiency, both in the programs produced and in the synthesis process itself. The efficiency problem arises because many target programs, which vary in their time and space performance, typically satisfy one abstract specification. There have been extensive works to devise several sophisticated techniques to address this problem, and the main drawback of current techniques is their inflexibility.

They work beautifully within their domain of expertise and not at all even a fraction outside of it. The only way to stretch a program generator beyond its intended domain is to manually modify the code produced by the generator. However, this is undesirable for two reasons. First, with most generators, the code created is intended only for compilation and is nearly unreadable. Modification of this kind of code is extremely error-prone. Second, a key benefit of program generators is that the output program can be modified to keep up with changing requirements by simply modifying the high-level input and rerunning the generator. Unfortunately, once the output of the generator has been manually modified, it has to be remodified each time the generator is rerun.

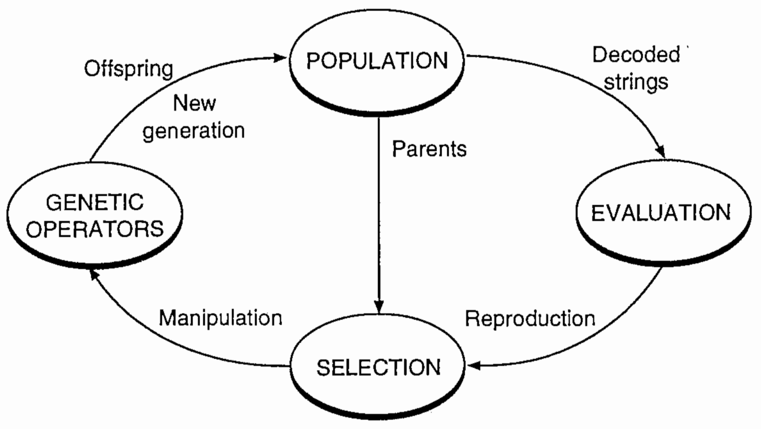

Figure 1: A typical procedure for evolutionary computation.

As a possible approach to solve the problem, this paper presents an evolutionary program transformation system and briefly discusses the merits of this approach. The methodology in detail can be found in Ray [7, 8, 9], and the software used is available over the network or on disk (see Appendix A). This section describes some of the basic mechanisms of the evolutionary computation with the genetic algorithm (GA).

Evolution is a remarkable problem-solving machine [10]. First proposed by John Holland in 1975, GAs as one of computational implementations are an attractive class of computational models that mimic natural evolution to solve problems in a wide variety of domains. A genetic algorithm emulates biological evolutionary theories to solve optimization problems.

A GA comprises a set of individual elements (the population) and a set of biologically inspired operators defined over the population itself. According to evolutionary theories, only the most suited elements in a population are likely to survive and generate offspring, thus transmitting their biological heredity to new generations. In computing terms, a genetic algorithm maps a problem onto a set of strings, each string representing a potential solution. The GA then manipulates the most promising strings in its search for improved solutions. A GA operates through a simple cycle of stages:.

1. creation of a *population* of strings,
2. evaluation of each string,
3. selection of *best* strings, and
4. genetic manipulation to create the new population of strings.

Figure 1 shows these four stages using the biologically inspired terminology. The evolutionary approach to program transformation takes pieces of programs as such strings.

2 System Structure

The work described here takes place on a virtual computer known as Tierra (Spanish for Earth). Tierra is a simulator for parallel computer of the MIMD (multiple instruction,

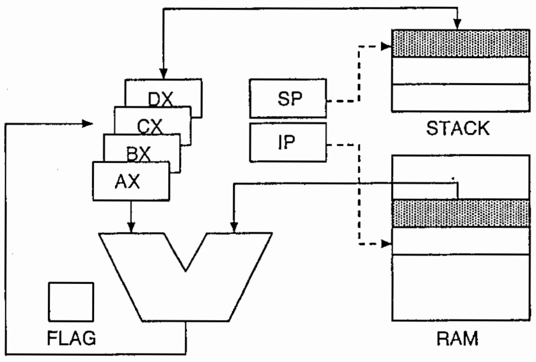

Figure 2: Structure definition to implement the Tierra virtual CPU.

multiple data) type, with a processor for each piece of program. Parallelism is imperfectly emulated by allowing each CPU to execute a small time slice in turn. Each CPU of this virtual computer contains two address registers, two numeric registers, a flag register to indicate error conditions, a stack pointer, a ten word stack, and an instruction pointer. Figure 2 shows the structure of each virtual CPU. Computations performed by the Tierran CPUs are probabilistic due to flaws that occur at a low frequency (see Section 2.2.4).

The instruction set of a CPU typically performs simple arithmetic operations or bit manipulations, within the small set of registers contained in the CPU. Some instructions move data between the registers in the CPU, or between the CPU registers and the RAM (main) memory. Other instructions control the location and movement of an instruction pointer (IP). The IP indicates an address in RAM, where the machine code of the executing program is located.

The CPU perpetually performs a fetch-decode-execute-increment_IP cycle: The machine code instruction currently addressed by the IP is fetched into the CPU, its bit pattern is decoded to determine which instruction it corresponds to, and the instruction is executed. Then the IP is incremented to point sequentially to the next position in RAM, from which the next instruction will be fetched. However, some instructions like JMP, CALL and RET directly manipulate the IP, causing execution to jump to some other sequence of instructions in the RAM.

2.1 Languages and formalism used

Before attempting to set up a program synthesis system, careful thought must be given to how the representation of a programming language affects its adaptability in the sense of being robust to genetic operations such as mutation and recombination. The nature of the virtual computer is defined in large part by the instruction set of its machine language. The approach in this study has been to loosen up the machine code in a *virtual biocomputer*, in order to create a computational system based on a hybrid between biological and classical von Neumann processes.

In developing this new virtual language, which is called *Tierran*, close attention has been paid to the structural and functional properties of the informational system of biological molecules: DNA, RNA and proteins. Two features have been borrowed from the biological world which are considered to be critical to the evolvability of the Tierran language.

First, the instruction set of the Tierran language has been defined to be of a size that is the same order of magnitude as the genetic code. Information is encoded into DNA through 64 codons, which are translated into 20 amino acids. In its present manifestation, the Tierran language consists of 32 instructions, which can be represented by five bits, *operands included.*

Emphasis is placed on this last point because some instruction sets are deceptively small. For example, RISC machines may have only a few opcodes, but they probably all use 32 bit instructions, so from a mutational point of view, they really have 2^{32} instructions. Inclusion of numeric operands will make any instruction set extremely large in comparison to the genetic code.

In order to make a machine code with a truly small instruction set, we must eliminate numeric operands. This can be accomplished by allowing the CPU registers and the stack to be the only operands of the instructions. When we need to encode an integer for some purpose, we can create it in a numeric register through bit manipulations: flipping the low order bit and shifting left. The program can contain the proper sequence of bit flipping and shifting instructions to synthesize the desired number, and the instruction set need not include all possible integers.

A second feature that has been borrowed from molecular biology in the design of the Tierran language is the addressing mode, which is called *address by template.* This is illustrated by the Tierran JMP (jump) instruction. Each JMP instruction is followed by a sequence of NOP (no-operation) instructions, of which there are two kinds: NOP_0 and NOP_1. Suppose we have a piece of code with five instruction in the following order: JMP NOP_0 NOP_0 NOP_0 NOP_1. The system will search outward in both directions from the JMP instruction looking for the nearest occurrence of the complementary pattern: NOP_1 NOP_1 NOP_1 NOP_0. If the pattern is found, the instruction pointer will move to the end of the complementary pattern and resume execution. If the pattern is not found, an error condition (flag) will be set and the JMP instruction will be ignored.

The Tierran language is characterized by two unique features: a truly small instruction set without numeric operands, and addressing by template. Otherwise, the language consists of familiar instructions typical of most machine languages, e.g., MOV, CALL, RET, POP, PUSH, etc. The complete instruction set is listed in Table 1.

2.2 Tierra simulator

The Tierra simulator needs a virtual operating system that will be hospitable to programs to change by evolution. The operating system will determine the mechanisms of interprocess communication, memory allocation, and the allocation of CPU time among competing processes. Algorithms will evolve so as to exploit these features to their advantage. More than being a mere aspect of the environment, the operating system together with the instruction set will determine the topology of possible interactions between individuals (see Figure 3).

2.2.1 Time sharing

The Tierran operating system must be multi-tasking (or parallel) in order for a community of individual programs to live in the workspace simultaneously. The system doles out small slices of CPU time to each program in the workspace in turn. The system maintains a

Table 1: Instruction set for the language used.

Category	Instruction	Operation
No operation	nop_0	no operation
	nop_1	no operation
Calculation	sub_ab	subtract bx from ax, cx = ax − bx
	sub_ac	subtract cx from ax, ax = ax − cx
	inc_a	increment ax, ax = ax + 1
	inc_b	increment bx, bx = bx + 1
	dec_c	decrement cx, cx = cx − 1
	inc_c	increment cx, cx = cx + 1
	zero	set cx register to zero, cx = 0
	or1	flip low order bit of cx, cx $\neq$ 1
	shl	shift left cx register, cx $\ll=$ 1
Memory management	push_ax	push ax on stack
	push_bx	push bx on stack
	push_cx	push cx on stack
	push_dx	push dx on stack
	pop_ax	pop top of stack into ax
	pop_bx	pop top of stack into bx
	pop_cx	pop top of stack into cx
	pop_dx	pop top of stack into dx
	mov_cd	move cx to dx, dx = cx
	mov_ab	move ax to bx, bx = ax
	mov_iab	move instruction at address in bx to ax
IP manipulation	if_cz	if (cx==0) execute next instruction
	jmp	move ip to template
	jmpb	move ip backward to template
	call	call a procedure
	ret	return from a procedure
Biological & Sensory	adr	address of nearest template to ax
	adrb	search backward for template
	adrf	search forward for template
	mal	allocate memory for daughter cell
	divide	cell division

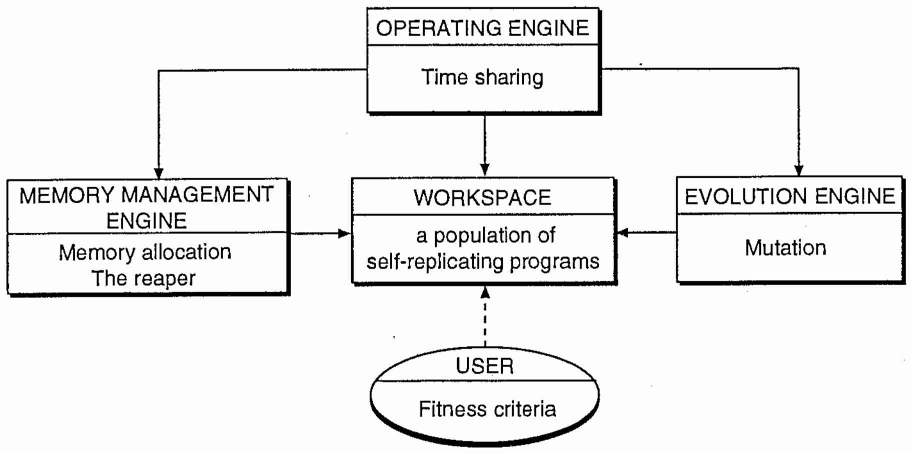

Figure 3: Overview of the system framework.

circular queue called the *slicer queue*. As each program is born, a virtual CPU is created for it, and it enters the slicer queue just ahead of its mother, which is the active program at that time. Thus the newborn will be the last program in the workspace to get another time slice after the mother, and the mother will get the next slice after its daughter. As long as the slice size is small relative to the generation time of the programs, the time sharing system causes the world to approximate parallelism. In actuality, we have a population of virtual CPUs, each of which gets a slice of the real CPU's time as it comes up in the queue.

The number of instructions to be executed in each time slice may be set proportional to the size of the program being executed, raised to a power. If the *slicer power* is equal to one, then the slicer is size neutral, the probability of an instruction being executed does not depend on the size of the program in which it occurs. If the power is greater than one, large programs get more CPU cycles per instruction than small programs. If the power is less than one, small programs get more CPU cycles per instruction. The power determines if selection favors large or small programs, or is size neutral. A constant slice size selects for small programs.

2.2.2 Memory allocation

The Tierran computer operates on a block of RAM of the real computer which is set aside for the purpose. This block of RAM is referred to as the workspace. In most of the work described here the workspace consisted of about 60,000 bytes, which can hold the same number of Tierran machine instructions. Each program occupies some block of memory in this workspace.

The Tierran operating system provides memory allocation services. Each program has exclusive write privileges within its allocated block of memory. This *membrane* is described as *semi-permeable* because while write privileges are protected, read and execute privileges are not. A program may examine the code of another program, and even execute it, but it cannot write it over. Each program may have exclusive write privileges in at most two blocks of memory: the one that it is born with which is referred to as the *mother cell*, and a second block which it may obtain through the execution of the MAL (memory allocation) instruction. The second block, referred to as the *daughter cell*, may be used to grow or reproduce into.

When Tierran programs *divide*, the mother cell loses write privileges on the space of the daughter cell, but is then free to allocate another block of memory. At the moment of division, the daughter cell is given its own instruction pointer, and is free to allocate its own second block of memory.

2.2.3 The reaper

Evolving programs in a fixed size workspace would rapidly fill the space and lock up the system. To prevent this from occurring, it is necessary to include mortality. The Tierran operating system includes a *reaper* which begins *killing* programs from a queue when the memory fills to some specified level. Programs are killed by deallocating their memory, and removing them from both the reaper and slicer queues. Their *dead* code is not removed from the workspace.

In the present system, the reaper uses a linear queue. When a program is born it enters the rear of the queue. The reaper always kills the program at the front of the queue. However, individuals may move ahead or back in the reaper queue according to their success or failure at executing certain instructions. When a program executes an instruction that generates an error condition, it moves one position ahead in the queue, as long as the individual ahead of it in the queue has not accumulated a greater number of errors. Two of the instructions are somewhat difficult to execute without generating an error, therefore successful execution of these instructions moves the program back in the reaper queue one position, as long as it has not accumulated more errors than the program below it. The effect of the reaper queue is to cause algorithms which are fundamentally flawed to rise to the top of the queue and die. Vigorous algorithms have a greater longevity, but in general, the probability of death increases with age.

2.2.4 Mutation

In order for evolution to occur, there must be some change in the program being generated. This may occur within the lifespan of an individual, or there may be errors in passing along the program to offspring. In order to insure that there is genetic change, the operating system randomly flips bits in the workspace, and the instructions of the Tierran language are imperfectly executed.

Mutations occur in two circumstances. At some background rate, bits are randomly selected from the entire workspace and flipped. This is analogous to mutations caused by cosmic rays, and has the effect of preventing any program from being immortal, as it will eventually mutate to death. The background mutation rate has generally been set at about one bit flipped for every 10,000 Tierran instructions executed by the system. In addition, while copying instructions during the replication of programs, bits are randomly flipped at some rate in the copies. The copy mutation rate is the higher of the two, and results in replication errors. The copy mutation rate has generally been set at about one bit flipped for every 1,000 to 2,500 instructions moved. In both classes of mutation, the interval between mutations varies randomly within a certain range to avoid possible periodic effects.

In addition to mutations, the execution of Tierran instructions is flawed at a low rate. For most of the 32 instructions, the result is off by plus or minus one at some low frequency.

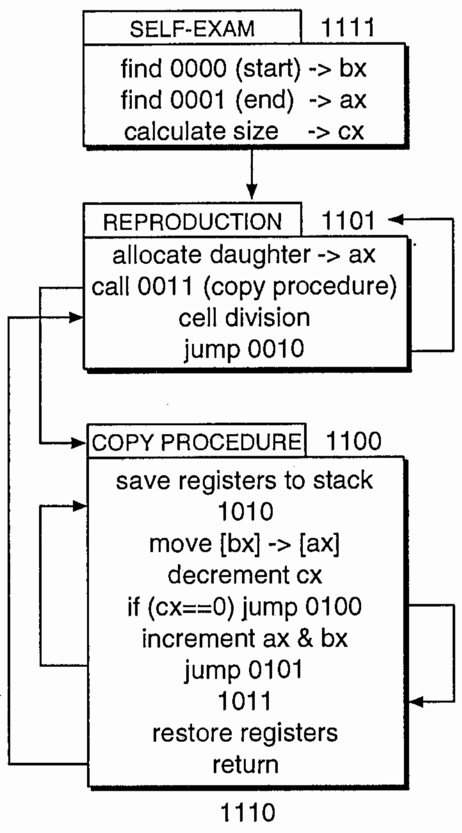

Figure 4: A flow chart for the ancestor program.

For example, the increment instruction normally adds one to its register, but it sometimes adds two or zero. The bit flipping instruction normally flips the low order bit, but it sometimes flips the next higher bit or no bit. The shift left instruction normally shifts all bits one bit to the left, but it sometimes shifts left by two bits, or not at all. In this way, the behavior of the Tierran instructions is probabilistic, not fully deterministic.

It turns out that bit flipping mutations and flaws in instructions are not necessary to generate genetic change and evolution, once the community reaches a certain state of complexity. Genetic parasites evolve which are sloppy replicators, and have the effect of moving pieces of code around between pieces of programs, causing rather massive rearrangements of the program.

3 A Case Study

3.1 Self-replicating program

We have used the Tierran language to write a single self-replicating program which is 80 instructions long. This program is referred to as the "ancestor," and Figure 4 illustrates how it looks. The ancestor is a minimal self-replicating algorithm, and no functionality was designed into the ancestor beyond the ability to self-replicate, nor was any specific evolutionary potential designed in.

The ancestor examines itself to determine where in memory it begins and ends. The ancestor's beginning is marked with the four no-operation template: 1 1 1 1, and its ending is marked with 1 1 1 0. The ancestor locates its beginning with the five instructions: ADRB, NOP_0, NOP_0, NOP_0, NOP_0. This series of instructions causes the system to search backwards from the ADRB instruction for a template complementary to the four NOP_0 instructions, and to place the address of the complementary template (the beginning) in the bx register of the CPU. A similar method is used to locate the end.

Having determined the address of its beginning and its end, it subtracts the two to calculate its size, and allocates a block of memory of this size for a daughter cell. It then calls the copy procedure which copies the entire program into the daughter cell memory, one instruction at a time. The beginning of the copy procedure is marked by the four no-operation template: 1 1 0 0. Therefore the call to the copy procedure is accomplished with the five instructions: CALL, NOP_0, NOP_0, NOP_1, NOP_1. When the program has been copied, it executes the DIVIDE instruction, which causes the program to lose write privileges on the daughter cell memory, and gives an instruction pointer to the daughter cell (it also enters the daughter cell into the slicer and reaper queues). After this first replication, the mother cell does not examine itself again; it proceeds directly to the allocation of another daughter cell, then the copy procedure is followed by cell division, in an endless loop.

Forty-eight of the eighty instructions in the ancestor are no-operations. Groups of four no-operation instructions are used as complementary templates to mark six sites for internal addressing, so that the program can locate its beginning and end, call the copy procedure, and mark addresses for loops and jumps in the code, etc.

3.2 Results

Evolutionary runs of the simulator are begun by inoculating the workspace of about 60,000 instructions with a single individual of the 80 instruction ancestral genotype. The passage of time in a run is measured in terms of how many Tierran instructions have been executed by the simulator. The original ancestral cell executes 839 instructions in its first replication, and 813 for each additional replication. The initial cell and its replicating daughters rapidly fill the workspace to the threshold level of 80% which starts the reaper. Typically, the system executes about 400,000 instructions in filling up the workspace with about 375 individuals of size 80. Once the reaper begins, the memory remains filled with programs for the remainder of the run.

The efficiency of the program generated can be indexed in two ways: the size of the program, and the number of CPU cycles needed to execute one replication. Clearly, smaller programs can be replicated with less CPU time, however, during evolution, programs also decrease the ratio of instructions executed in one replication, to program size. The number of instructions executed per instruction copied, drops substantially. The smallest limiting program size seen has been 22 instructions (see Figure 5).

The increase in efficiency of the replicating algorithms is even greater than the decrease in the size of the code. The ancestor is 80 instructions long and requires 839 CPU cycles to replicate. The program of size 22 only requires 146 CPU cycles to replicate, a 5.75–fold difference in efficiency.

Programs present at the end of some runs were examined and found to have evolved an intricate adaptation. The adaptation is an optimization technique known as *unrolling*

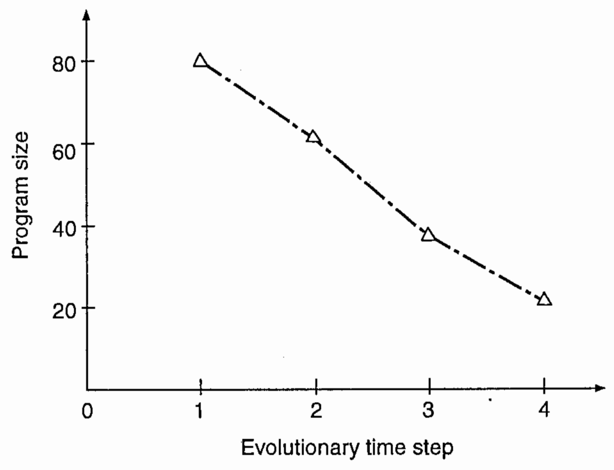

Figure 5: A comparison of optimization of program size as the evolution goes on.

the loop. The central loop of the copy procedure performs the following operations: 1) copies an instruction from the mother to the daughter, 2) decrements the cx register which initially contains the size of the parent program, 3) tests to see if cx is equal to zero, if so it exits the loop, if not it remains in the loop, 4) increment the ax register which contains the address in the daughter where the next instruction will be copied to, 5) increment the bx register which contains the address in the mother where the next instruction will be copied from, 6) jump back to the top of the loop.

The work of the loop is contained in steps 1, 2, 4 and 5. Steps 3 and 6 are overhead. The efficiency of the loop can be increased by duplicating the work steps within the loop, thereby saving on overhead. Programs from the end of some long runs had repeated the work steps two or three times within the loop.

Unrolling of the loop results in a loop which uses 18 CPU cycles to copy three instructions, or six CPU cycles executed per instruction copied, compared to 10 for the ancestor. The program of size 22 also uses six CPU cycles per instruction copied. However, the program exploiting the unrolling the loop uses three extra CPU cycles per loop to compensate for a separate adaptation that allows it to double its share of CPU time from the global pool (in essence meaning that relatively speaking, it uses only three CPU cycles per instruction copied). Without this compensation it would use only five CPU cycles per instruction copied.

3.3 Discussion

A wide variety of programming systems make use of program transformations as part of their knowledge representation and automation apparatus. They use transformations to improve the efficiency of a program. The input to the system is a clear but inefficient program; the output is a program that is much more efficient but usually much less clear. The system operates by applying a small number of powerful transformations to merge information from different parts of the input program and then redistribute it into a more efficient modularization.

Obviously, it is necessary to point out a couple of problems in the program transformation and synthesis and how the evolutionary approach works them out. In the use of transformations, it is important to be able to verify that the transformations are correctness-preserving, i.e., that the transformation process does not alter the intended behavior of a program. The evolutionary approach guarantees the property with the appropriate criteria given for selection process. This means that the user is only required to supply the fitness criteria instead of the sophisticated transformation rules.

Another paramount problem in using program transformations is choosing which transformation to apply in situations where several are applicable. This requires the system to have some control structures that allow it to search all possible transformational paths, creating a variety of output programs. Besides, for each new application, the user is encouraged to define syntactic and semantic extensions of the transformation rules that capture the natural control structures and data abstractions of the application area. In the evolutionary approach, however, there is no need for the user to specify the transformation knowledge manually. Evolution process might give a way to automatically find the efficient rules.

4 Concluding remarks

Efficient program synthesis is a problem that draws on program synthesis technology and analysis of algorithms. As an exercise in program synthesis, the evolutionary framework offers a focus on automatically guiding the production of efficient programs. It allows global optimizations based on an overall view of the uses through evolutionary process. In this paper, we have described a preliminary system based on the framework.

We hope that evolutionary techniques for program transformation can be utilized to provide a practical programming tool and have started the modification of systems to accomplish this. These systems will put more control back with the user requiring him/her to have an intuitive idea of the development of his/her algorithm leaving to the system of task of implementing this development correctly.

Acknowledgements

The authors would like to thank Dr. K. Shimohara and Dr. Y. Tohkura at ATR HIP laboratories for continuous encouragement.

Appendix A. Getting the Tierra system

The complete source code and documentation is available by anonymous ftp at:
`tierra.slhs.udel.edu [128.175.41.34]` or
`life.slhs.udel.edu [128.175.41.33]`
as the file `tierra/tierra.tar.Z`. The source code compiles and runs on either DOS or UNIX systems. If you do not have ftp access, the complete UNIX/DOS system is also available on DOS disks with an easy installation program. For the disk set, contact the second author.

References

[1] L. A. Rowe and F. M. Tonge, "Automating the selection of implementation structures", *IEEE Trans. on Software Eng.* **SE-4** (1978) 494–506.

[2] E. Schonberg, J. T. Schwartz and M. Sharir, "An automatic technique for selection of data representations in SETL programs", *ACM Trans. on Prog. Lang. and Sys.* **3** (1981) 126–143.

[3] Z. Manna and R. Waldinger, "A deductive approach to program synthesis", *ACM Trans. on Prog. Lang. and Sys.* **2** (1980) 90–121.

[4] R. C. Waters, "The programmer's apprentice: a session with KBEmacs", *IEEE Trans. on Software Eng.* **SE-11** (1985) 1296–1320.

[5] M. Broy and P. Pepper, "Program development as a formal activity", *IEEE Trans. on Software Eng.* **SE-7** (1981) 14–22.

[6] T. E. Cheatham, Jr., "Reusability through program transformations", *IEEE Trans. on Software Eng.* **SE-10** (1984) 589–594.

[7] T. S. Ray, "An approach to the synthesis of life", *Artificial Life II*, Santa Fe Institute Studies in the Sciences of Complexity, **2**, CA: Addison-Wesley, 1991, pp. 371–408.

[8] T. S. Ray, "Evolution, complexity, entropy, and artificial reality", *Physica D* **75** (1994) 239–269.

[9] T. S. Ray, "An evolutionary approach to synthetic biology: zen and the art of creating life", *Artificial Life* **1** (1994) 179–209.

[10] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey", *IEEE Computer*, June 1994, 17–26.