数理モデル化と問題解決　27－3
（1999．11．25）

コアメモリ型人工生命システムでオープンエンドな進化を誘発するための条件

鈴木秀明　　トーマス S. レイ†
（株）ＡＴＲ 人間情報通信研究所

コアメモリ型人工生命システムでオープンエンドな進化を誘発させるためには、その機能ユニットであるタンパク質の進化能力をできるだけ高めるようにシステムを設計する必要がある。本論文では、ひとつながりのデジタルアミノ酸によって構成されるデジタルタンパク質を考え、その進化能力を高めるための４つのデザイン上の規範を与える。さらに事例研究としてSeMarと呼ぶ人工生命システムを対象にし、与えた条件をもとにシステムの基本デザイン（レジスタ構成と基本命令セット）を最適化する例を示す。

Several Conditions to Cause Open-ended Evolution in Core-memory-type ALife Sytems

Hideaki Suzuki　　Thomas S. Ray†
ATR Human Information Processing Research Laboratories

A set of conditions is presented to enhance the evolvability of an artificial core memory programming system. A digital protein (CPU) composed of a sequence of digital amino acids (instructions) is considered, and a set of implementation criteria is given to maximize the functionality and evolvability of proteins. Next, for a case study, we present the basic design of an ALife system named SeMar. The design of SeMar proteins (elementary instructions and a register architecture) is optimized considering the above conditions.

1 Introduction

Evolution is a transition from one self-reproducing genotype to another self-reproducing genotype. If the two genotypes are located apart in the genotype space, the transition can never happen in an appropriate waiting time, so for one species to evolve from another, both species must be connected by a sequence of species (genotypes) which are self-reproducing and slightly different from each other.

The distribution of self-reproducing genotypes in the genotype space influences the ability of a system to evolve a variety of organisms. Because mutation can only slightly modify the genotypes, if the self-reproducing genotypes are sparsely distributed and isolated, mutation cannot easily cause changes from one isolated region to another. On the other hand, if the self-reproducing genotypes are densely distributed and interconnected, mutation can easily explore very widely through the genotype space and the system can evolve a variety of self-reproducing genotypes. Accordingly, we can discuss system "evolvability" (see [1] for example) if we can estimate the density of self-reproducing genotypes.

This argument is true of an ALife (artificial life) system that evolves artificial creatures as well. In the Tierra system [2, 3], an ancestor creature is coded in a 400-bit string, and a parasite can be made to evolve from this ancestor by flipping only one bit in the string. This suggests that in Tierra, self-reproducing genotypes are densely distributed, and an ancestor genotype and a parasite genotype are mapped close to each other in the genotype space. To make an ALife system highly evolvable, we must optimize the design so that this density might be maximized.

Although testing the self-reproductivity of an entire creature is a difficult task [7], if we focus on the genotype of a protein, we can estimate evolvability more easily. Proteins are functional units in the body of an organism, and their high evolvability (the ability for proteins to evolve various functions) has evidently contributed to the evolvability of the organism. The evolvability of proteins is largely dependent upon their "functionality" (the ability for polypeptides to have functions); hence, if functional proteins are densely distributed in the protein genotype space, the system will be highly evolvable. Based on this notion, here we propose a set of conditions to facilitate the functionality of digital proteins. Although these conditions are

principally derived from observations (or spec tions) on living proteins, they greatly help o mize the design of ALife systems, which are pected to have open-ended evolution.

In the following, first, we present four conditi to facilitate the functionality of digital prot (Sect. 2). Then, taking SeMar [5, 6, 7] as an ample, we give several instances of protein de optimization based on these conditions (Sect. Conclusion is given in Sect. 4.

2 Conditions to Enhan Evolvability

Here, we assume that a digital protein is coded binary string that codes a sequence of instructi (digital amino acids). The final function of s a digital protein is determined by the execut of these instructions. Conditions to enhance functionality of proteins of this type are given follows.

Cond. 1: The number of mutable bits with possibility to influence the functionality of protein should be kept to a minimum.

Cond. 2: When instructions (or bits) are randomly placed in a sequence, the sequence must be functional as much as possible no matter how long/short the sequence is.

Cond. 3: If a sequence is divided into two subsequences with appropriate lengths, each of the two subsequences must be functional and have component functions of the entire function as much as possible. Or, on the contrary, if two different sequences are connected to make a single sequence, the combined sequence must have the combined functionality of the two functions as much as possible.

Cond. 4: Operand data accessed by an instruction must be specified as implicitly as possible.

3 Case Study

In this section we describe the design of a digital protein in SeMar, and gives its characteristics referring to Conds. 1∼4.

3.1 The Model of SeMar

SeMar (the Sea of Matter) [5, 6, 7] is an evolutionary programming system which uses a dynamic core memory. Figure 1 shows a snapshot of a part of the SeMar core. Every word in the core

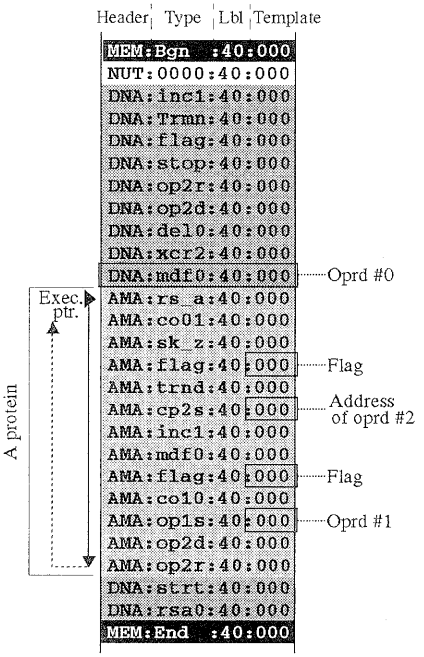

Figure 1: A snapshot of a part of the SeMar core

is coded in a 48-bit string and denoted by punctuated mnemonics (called 'Header', 'Type', 'Label', and 'Template' in turn). Header is either `MEM`, `NUT`, `DNA`, or `AMA` and denotes the kind of word. Type represents the kind of codon/amino acid for a `DNA`/`AMA` word, or the kind of type (`Bgn` or `End`) for a `MEM` word. Label holds a string that is peculiar to each word. Template is used for matching with other words' labels and helps select operand data.

A digital protein of SeMar (a consecutive set of `AMA` words) can be compared to the CPU of a computer. It can store four kinds of registers: oprd #0 is the word in front of the protein, and the other three registers (#1, #2, and flag) can be present plurally in the protein (Fig. 1). At each clock, a protein executes its inner program (the codon sequence) sequentially and cyclically, and after the execution, it takes one step forward (upward in Fig. 1). If the protein bumps into `MEM:Bgn` data, it is immediately transfered to before the corresponding `MEM:End` word. At each clock, all proteins in the core are put into action sequentially; and yet the time resource given to a protein per clock is so small that we can consider the actions of the proteins to be practically parallel.

Table 1 shows the prepared 30 elementary in-

Type	Temp	a=0	Operation
flag	(U)		No execution (Contains flags (a, p, n, and z)).
op1s	(U)		No execution (Contains oprd #1).
cp2s	(U)		No execution (Contains the address of oprd #2).
trnd	U		Searches the next nearest oprd #0 using Template, moves Protein after it, and disappears.
trnr	U		Searches the next nearest oprd #0 using Template and moves Protein after it.
op2d	U	N	Searches the oprd #2 nearest the current oprd #2 address (the address stored in the nearest AMA:cp2s) using Template, stores its address into an AMA:cp2s word, and disappears. If there is no AMA:cp2s word in the neighborhood, it creates a new one.
op2r	U	N	Searches the oprd #2 nearest the current oprd #2 address (the address stored in the nearest AMA:cp2s) using Template and stores its address into an AMA:cp2s word. If there is no AMA:cp2s word in the neighborhood, it creates a new one.
del0	D	N	Deletes oprd #0.
mdf0	(U)	N	Modifies oprd #0 using the Template data.
cre0	D	N	Inserts a copy of oprd #0 after Protein.
dec1	D	N	Decreases oprd #1 by one.
inc1	D	N	Increases oprd #1 by one.
co01	D	N	Copies oprd #0 to oprd #1. If there is no AMA:op1s word in the neighborhood, it creates a new one.
co10	D	N	Copies oprd #1 to oprd #0.
co02	D	N	Copies oprd #0 to oprd #2.
co20	D	N	Copies oprd #2 to oprd #0.
cr02	D	N	Inserts a copy of oprd #0 after oprd #2.
xc02	D	N	Exchanges oprd #0 for oprd #2.
rs_a	D		Sets flag 'a' in the nearest AMA:flag to 0.
st_a	D		Sets flag 'a' in the nearest AMA:flag to 1.
rsa0	U		Sets flag 'a' in the nearest AMA:flag to 0 if oprd #0 matches Template.
sta0	U		Sets flag 'a' in the nearest AMA:flag to 1 if oprd #0 matches Template.
rs_f	D		Sets flag data (p, n, and z) in the nearest AMA:flag to 0.
ts_1	D		Sets flag data (p, n, and z) of oprd #1 into the nearest AMA:flag.
ts02	D		Sets flag data (p, n, and z) of (oprd #0−o 爵 prd #2) into the nearest AMA:flag.
die0	U		Deletes all of Protein except for the stored data (AMA:flag, op1s, and cp2s) if oprd #0 matches Template.
sk_a	D		Skips the next instruction if a=0 in the nearest AMA:flag.
sk_p	D		Skips the next instruction if p=0 in the nearest AMA:flag.
sk_n	D		Skips the next instruction if n=0 in the nearest AMA:flag.
sk_z	D		Skips the next instruction if z=0 in the nearest AMA:flag.

Table 1: The elementary instruction set (digital amino acid set). 'U' or 'D' in the 'Temp' column means that the execution uses or does not use the Template data in the word. '(U)' in the same column means that the Template field is used for storing the immediate data. 'N' in the 'a=0' column means that the instruction is not executed if all inner 'a' flags in the protein are 0.

structions (codons or amino acids). They are classified into four groups. The first group (`flag` to `cp2s`) is not executed and works as CPU registers manipulated by other executing instructions. The second group (`trnd` to `op2r`) includes instructions for determining oprds #0 and #2. The third group (`del0` to `xc02`) specifies operations among oprds #0, #1, and #2. The fourth group (`rs_a` to `sk_z`) is for manipulating `AMA:flag` data.

3.2 Characteristics of the Protein Design

The protein design given in the previous subsection was made considering Conds. 1~4 in Sect. 2. In the following, we summarize the characteristics of the design together with the corresponding conditions.

There is no supervisory or required instruction in a protein [Conds. 1 and 3]. Almost all instructions can be executed without other instructions, so that the possibility is small for the replacement of an instruction to make another instruction unfunctional.

Mutation can occur only on the bit fields of `DNA` words excluding the Header [Cond. 1]. The separation of a protein into two parts is most of the time deleterious. To prevent mutation from causing this separation, mutation is operated only on a limited region of the bit field. Specifically, mutation on the Header field is forbidden.

Working registers of a protein are buried into the core [Conds. 2 and 3]. Because most of a protein's (CPU's) operations are modifications of the contents of working registers, if these registers are prepared apart from the core (as they are in other ALife systems [2, 4]), the protein needs to finally execute a writing instruction to write the result on the core. The functionality (ability to modify the core data) of a protein is dependent on the execution of a few writing instructions in the protein, and if such a protein is divided into two parts, either of the two smaller proteins is very likely to lose its functionality. To avoid this problem, the protein registers in SeMar (oprds #0 ~ #2 and flag) are prepared as core words. This is expect to enhance the functionality of small proteins.

Registers necessary for the execution of an instruction are searched for beyond the border of the protein [Conds. 2 and 3]. The execution of an instruction needs certain operand registers. For example, the `co10` instruction manipulates oprd #1; hence, if an `AMA:op1s` word is not found at the time of execution, the instruction cannot be functional. To maximize the possibility of an instruction finding operand data, a SeMar instruction searches the core for registers not only in the inside of the protein but also in the neighborhood of the core. This not only enhances the functionality of the protein, of which the sequence is chosen randomly [Cond. 2], but also makes neighboring proteins cooperate with each other via common registers [Cond. 3].

The number of working registers is small (three, #0~#2), and operands are searched for using bit matching with Template [Cond. 4]. Like Tierra, SeMar specifies operand data using the bit matching between Label and Template. By adjusting the effective length of Template, the possibility that an instruction finds an operand can be enhanced appropriately.

4 Conclusion

Conditions to enhance the evolvability of digital proteins were presented and were applied to an ALife system named SeMar. It was shown that the conditions are useful in optimizing the design of proteins.

References

[1] Bedau, M. A., Snyder, E., Packard, N. H.: A Classification of Long-Term Evolutionary Dynamics. In: Adami, C. et al. (eds.): Artificial Life VI Proceedings. MIT Press, Cambridge (1998) 228-237

[2] Ray, T.S.: An approach to the synthesis of life. In: Langton, C.G. et al. (eds.): Artificial Life II Proceedings. Addison-Wesley (1992) 371–408

[3] Ray, T.S., Hart, J.: Evolution of differentiated multi-threaded digital organisms. In: Adami, C. et al. (eds.): Artificial Life VI Proceedings. MIT Press, Cambridge, MA (1998) 295–304

[4] Suzuki, H.: Multiple von Neumann Computers: An evolutionary approach to functional emergence. Artificial Life **3** (1997) 121–142

[5] Suzuki, H.: One-dimensional unicellular creatures evolved with genetic algorithms. In: JCIS '98: The Fourth Joint Conference on Information Sciences, Proceedings Vol.II. Association for Intelligent Machinery Inc., USA (1998) 411–414

[6] Suzuki, H.: A Simulation of Life Using a Dynamic Core Memory Partitioned by Membrane Data. In: Floreano, D. et al. (eds.): ECAL Porceedings. Springer-Verlag, Berlin (1999) 412-416

[7] Suzuki, H.: A Necessary Condition for Self-reproduction in the Semar Core. In: 1999 IEEE International Fuzzy Systems Conference Proceedings (1999) 123–128

[8] Yura, K., Go, M.: Helix-Turn-Helix Module Distribution and Module Shuffling. In: Go, M., Schimmel, P. (eds.). Tracing Biological Evolution in Protein and Gene Structures. Elseviers (1995) 187-195