# Hardware Evolution System Introducing Dominant and Recessive Heredity

Tomofumi Hikage[†], Hitoshi Hemmi[††], and Katsunori Shimohara[†]

[†]NTT Human Interface Laboratories
1-2356 Take Yokosuka-Shi
Kanagawa 238-03, JAPAN
[††]Evolutionary Systems Department,
ATR Human Information Processing Research Laboratories,
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-02, JAPAN
hikage@nttcvg.hil.ntt.jp, hemmi@hip.atr.co.jp, katsu@nttcvg.hil.ntt.jp

**Abstract.** This paper proposes a new hardware evolution system – a new AdAM (Adaptive Architecture Methodology), that introduces dominant and recessive heredity through diploid chromosomes in order to increase genetic diversity. Dominant and recessive heredity is implemented by two techniques: one node of a tree-structured chromosome can have two sub-trees corresponding to alleles: Dominant or recessive attributes of a new pair of sub-trees is decided randomly. Simulations using the artificial ant problem show that the new AdAM is superior to the old one in adaptability and robustness in the face of a changeable environment.

## 1 Introduction

This paper proposes a new AdAM (Adaptive Architecture Methodology) system that introduces genetic diversity to enhance system adaptability and robustness. In order to achieve genetic diversity, we graft dominant and recessive heredity onto the original AdAM [3].

Hardware evolution will enable a system to refine its hardware structure and behavior dynamically. Hardware evolution is now becoming a promising research field given recent progress in evolution methodology and reconfigurable hardware like the FPGA (Field Programmable Gate Array).

In order to create complex adaptive systems, hardware evolution and software evolution are indispensable. Hardware evolution is most effective when the designer can't grasp all conditions at the design stage. In particular, hardware evolution or evolvable hardware will greatly advance of the realization of autonomous robots suitable for uncertain and fluctuating environments.

One approach is to directly treat the configuration data of the FPGA as chromosomes and evolve them using the Genetic Algorithm [2]. Unfortunately, since the configuration data is low-level, it is difficult to apply this approach to large-scale and complex applications.

We are aiming to establish an evolutionary methodology for hardware and to create a computational framework to guide evolutionary hardware. For that

purpose, we employ HDL (Hardware Description Language), which provides the new AdAM with the same advantages in terms of both software and hardware high-level description and flexibility, as it does the original AdAM.

HDL programs, which are converted into hardware behavior, are automatically generated and forced to evolve. In the sense that programs evolve as chromosomes, AdAM can be regarded as a form of software evolution. However, we can deal with hardware behavior and structure as pheno-types by using HDL-based programs. Emphasis should be paid to the fact that we treat hardware the same as software.

Dominant and recessive heredity is part of the new AdAM to increase genetic diversity, so that the system can achieve much more adaptability and robustness in changeable and fluctuating environments. To implement dominant and recessive heredity, nodes of tree-structured chromosome are extended so as to have a pair of sub-trees which represent rewriting rules as alleles. Since this creates diploids, only one node of every chromosome is randomly selected to have a pair of sub-trees, and genetic operations are controlled to generate only diploids. Whether a rewriting rule is dominant or recessive is decided randomly as each rule pair appears.

In this paper, we compare the new AdAM to the old one to verify adaptability and robustness in changeable and fluctuating environments through simulations on the artificial ant problem.

## 2 How to Model Dominant and Recessive Heredity

Most multi-cellular organisms are diploid with a set of diploid chromosomes, and exhibit dominant and recessive heredity; a haploid, by comparison, has only one set of chromosomes. Since we usually employ a model in which one individual has one "chromosome", we postulate that a diploid has a pair of chromosomes and a haploid has a chromosome to simplify the following discussion.

In the case of a haploid, a change in a chromosome immediately appears as a change in its pheno-type experiencing selection pressure. In general, most changes in a chromosome due to mutation or deletion are harmful, in the sense that the probability of destroying a gene that generates an effective function is higher than that of gaining a new gene with a superior function. In order for a haploid to acquire a new function effectively, gene duplication should occur before having mutation. Such duplication allows the haploid to have copies of a gene for an existing function in pheno-type, its chromosome can acquire a new gene by mutation while keeping the original gene. That is, gene duplication must precede mutation so haploid evolution takes a long time in general.

Genetic Programming (GP) and genetic algorithms (GA) usually employ a haploid model. This means that GP and GA have the same drawback mentioned above. In GP and GA, however, a crossover operation is introduced, with which two different chromosomes, i.e., individuals, exchange a part of their chromosomes to offset the drawback.

A diploid, on the other hand, has a pair of chromosomes that are different in most cases. One gene is, in pheno-type, dominant. Thus, when it is changed by mutation, its pheno-type suffers selection pressure according to its fitness for the environment. The other gene, the recessive one, does not appear in the pheno-type and can be changed irrespective of the pheno-type. That is, even if a change in a recessive gene yields an effective function, it does not immediately improve fitness.

A diploid does not need gene duplication before mutation, unlike a haploid, because the recessive gene is available even if the dominant one is destroyed by mutation. Moreover, if gene duplication occurs in either of the pair genes, and if crossover between the pair of chromosomes takes place, the dominant and recessive genes may be placed in the same chromosome. Thus, a diploid has much more diversity and a higher possibility of acquiring a new gene than a haploid, while still keeping the recessive gene [1].

A gene in a chromosome can be dominant or recessive, so a chromosome usually consists of a mix of dominant and recessive genes. However, system becomes too complex and complicated if we employ an equivalent chromosome structure. Thus, when we mimic diploid chromosomes and dominant-recessive heredity, chromosomes are randomly defined as dominant or recessive, for simplicity. In addition, since a chromosome in our system consists of a node tree as described later, we model diploid and dominant-recessive heredity with one chromosome, in which only one node has two sub-trees corresponding to alleles, instead of a pair of chromosomes.

## 3 New AdAM as an Autonomous Hardware Evolutionary System

### 3.1 Outline of the System

Figure 1 is the system block diagram of the new AdAM. It is the same as that of the old AdAM except chromosome expression and some genetic operations that reflect dominant-recessive heredity.

The HDL that we use is SFL (Structured Function description Language), developed for PARTHENON (Parallel Architecture Refiner THEorized by Ntt Original coNcept) system - a kind of CAD/CAM system for ASIC LSIs. AdAM is an evolutionary system based on the autonomous generation of HDL programs. SFL is a high-level language like C and so offers high-level descriptions that are easy to understand.

Production rules to generate chromosomes are a set of SFL grammar expressed by BNF (Backus-Naur Form) which work as rewriting rules. By applying such rewriting rules repeatedly, a chromosome is produced as a node tree in which each node corresponds to a rewriting rule. As a chromosome is expressed as a node tree, genetic operations can be assigned to the chromosome. Figure 2 shows a part of an SFL grammar definition (BNF expression). In Figure 2, the rX.m in round brackets expresses rule-number, where X is category-number

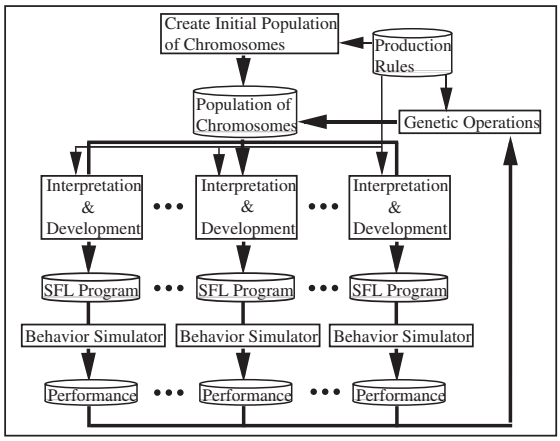


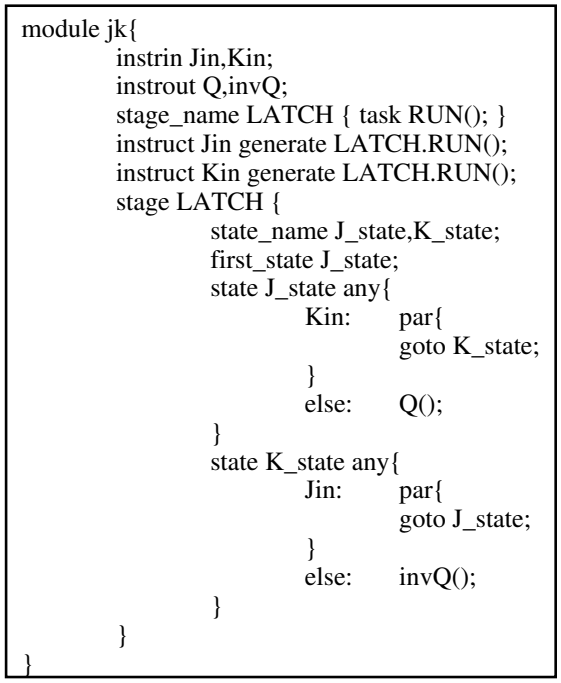

**Fig. 1.** System Block Diagram

```
(r0.0)  sfl_desc    ->    seq0_mod_def
(r1.0)  seq0_mod_def ->   empty
(r1.1)  seq0_mod_def ->   seq0_mod_def mod_def
(r2.0)  mod_def ->   K_MODULE mod_name K_LBRACE seq0_submod_type_OR_fc_type_def
          seq0_mod_component_def seq0_mod_ctrl_pin_arg_def seq0_stage_and_task_def
          seq0_mod_act_with_ctrl_pin_def seq0_stage_act_process_def K_RBRACE
(r3.0)  mod_name ->   nm_mod
                •
                •
                •
```

**Fig. 2.** A part of SFL grammar definition (BNF expression)

and m is sub-number. For example, r1.0 expresses rule-number of seq0_mod_def → empty, where 1 is category-number of seq0_mod_def and 0 is the sub-number of empty. This rule rewrites seq0_mod_def into empty. Equally, r1.1 expresses another rewriting rule of category seq0_mod_def.

A chromosome – a tree structure of rewriting rules based on SFL grammar – is automatically interpreted and developed into an SFL program that describes hardware behavior. Each non-terminal symbol, shown in small letters in Figure 2, has a rewriting rule. The capital letters in the figure are terminal symbols that have no rewriting rules. Non-terminal symbols must be rewritten as terminal symbols. Figure 3 indicates a simple SFL program.



```
module jk{
        instrin Jin,Kin;
        instrout Q,invQ;
        stage_name LATCH { task RUN(); }
        instruct Jin generate LATCH.RUN();
        instruct Kin generate LATCH.RUN();
        stage LATCH {
                state_name J_state,K_state;
                first_state J_state;
                state J_state any{
                        Kin:    par{
                                goto K_state;
                        }
                        else:   Q();
                }
                state K_state any{
                        Jin:    par{
                                goto J_state;
                        }
                        else:   invQ();
                }
        }
}
```

**Fig. 3.** Example of SFL program

This example program describes JK latch. A description of the module is that the description unit of SFL starts at K_MODULE(module) and finishes at K_RBRACE(}) according to BNF expression guidelines.

Overall system behavior is as follows: First, the initial chromosome population is randomly created according to the production rules. Each chromosome generates an individual SFL program through interpretation and development. Individual SFL programs, i.e., individual hardware behaviors, are simulated and evaluated by a given fitness function. Individuals with good performance are selected so that they become a parent of the next generation. Genetic operations are applied to the chromosomes of the parents to generate the chromosomes of the next generation. Such evolutionary processes eventually evolve individuals that encode for desirable hardware behaviors.

### 3.2 Chromosome with Dominant-Recessive Heredity

Figure 4 indicates a chromosome with dominant-recessive heredity.

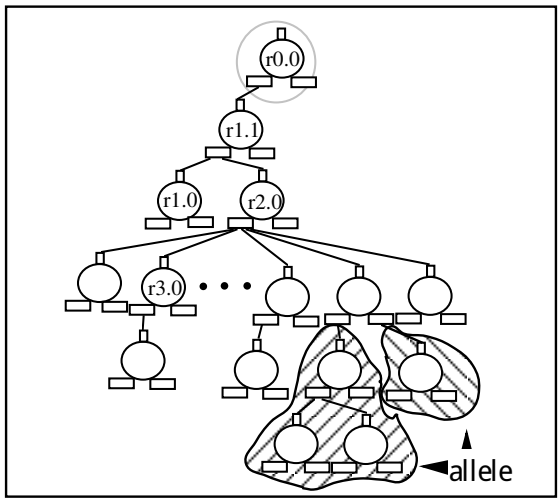



**Fig. 4.** A chromosome with dominant-recessive heredity

As mentioned before, we model dominant-recessive heredity with one chromosome by assigning one node to have two sub-trees corresponding to alleles. As shown in figure 4, a chromosome has a tree-structure of nodes each of which has a rule-number of a rewriting rule. In figure 4, two sub-trees are delineated by shading. One sub-tree is dominant; the remaining one is recessive.

Which node has two sub-trees is randomly determined when a chromosome is generated as follows: First, a node with rule-number r0.0 is generated. Category-number 0 expresses sfl_desc, and sfl_desc is rewritten to seq0_mod_def. The non-terminal symbol seq0_mod_def is assigned category-number 1 and has two sub-numbers 0 and 1. A node with a rule-number that has sub-numbers more than one, like seq0_mod_def, can be the node expressing dominant-recessive heredity, and the node used to express dominant-recessive heredity is randomly selected among all nodes with sub-number more than one. In the selected node, rules whose sub-numbers are randomly selected from those in the same category are allocated to two sub-trees of the node, and then which sub-tree is dominant is also determined randomly. If a node is not selected for dominant-recessive heredity, one of rules with sub-numbers is selected randomly. For example, a node next from r0.0 is generated by r1.0 or r1.1.

Moreover, especially by reproduction, one of genetic operations, described later, it happens that two sub-trees for alleles change to have both dominant or recessive. In the case, dominant or recessive of two sub-trees is also determined randomly.

### 3.3 Genetic Operations

Along with introducing dominant-recessive heredity into a chromosome, we define and employ the following genetic operations. These genetic operations have to be controlled so that chromosomes for diploids are always generated.

- Mutation: This operation changes the sub-number of a rule in a node, e.g., from rX.1 to rX.2 or rX.3, or changes the constant information of a node or the node information of constant number.
- Duplication / Deletion: This operation duplicates or deletes a node by changing the number of times the recursive rewriting rule is applied.
- HL (HapLoid)-type crossover: This operation exchanges the sub-numbers of two nodes, which have the same category-number, but each of which belongs to a different chromosome. As the only information exchanged is the sub-number of the same category number, each chromosome is kept grammatically correct. Figure 5 indicates the HL-type crossover operation.

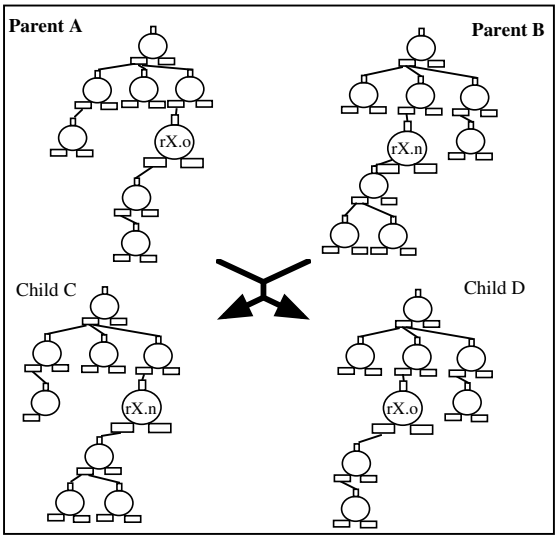



**Fig. 5.** Example of HL-type Crossover operation

- DL (DipLoid)-type crossover: This operation is newly introduced along with dominant-recessive heredity. It exchanges the sub-numbers of two nodes, which have the same category-number, but each of which belongs to either of two sub-trees corresponding to alleles in the same chromosome. Therefore, a part of recessive gene may be included in a dominant sub-tree, and vice versa.

– Reproduction: This operation is also newly introduced, and is the operation most characteristic of dominant-recessive heredity. It exchanges the sub-trees of two nodes that have the same category-number, but each of which belongs to a different chromosome. For example, the chromosomes owned by parent A and B yield child C and D, as shown in figure 6.

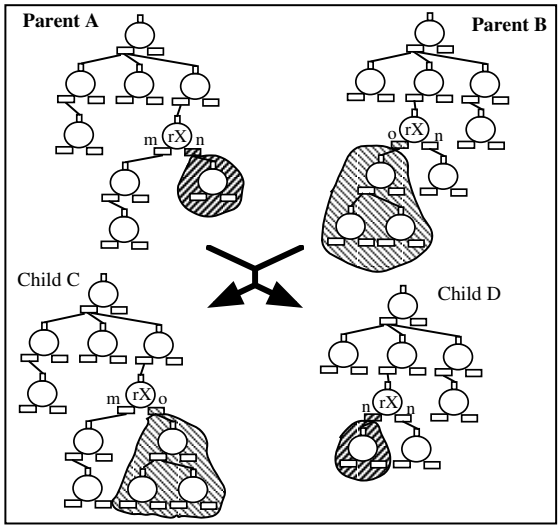



**Fig. 6.** Example of Reproduction with dominant and recessive heredity

Reproduction is to fuse two "germ cells", that is, the germ cells of parent A and B are fused to yield child C and child D. The dominant or recessive tags of the sub-trees of parent A and B are already determined before reproduction. In figure 6, X indicates category-number, and m, n and o indicate sub-numbers. If we assume that sub-number m and o are dominant for sub-number n, the sub-number n can not appear in pheno-type at the parent's generation. However, at the child's generation, child D has two recessive sub-numbers in its two sub-trees, that is, both sub-trees are recessive. In that case, the recessive sub-number appears in pheno-type.

The operations described above prevent any genetic operation from generating lethal chromosomes. Thus, efficient and effective evolution without waste can be achieved, and the SFL programs resulting from genetic operations can also satisfy the SFL program definition requirements.

## 4 Experiment

We compared the new AdAM to the old AdAM by examining their performance in terms of adaptability and robustness under environmental fluctuation; both were applied to the artificial ant problem. This problem is to evolve the hardware behavior of artificial ants so that they collect food faster and more effectively. The best artificial ant should collect the most food in the fewest steps. Figure 7 indicates the environment of the problem. The black squares are food. The experiment environment is a torus. The artificial ant starts at the top left corner.

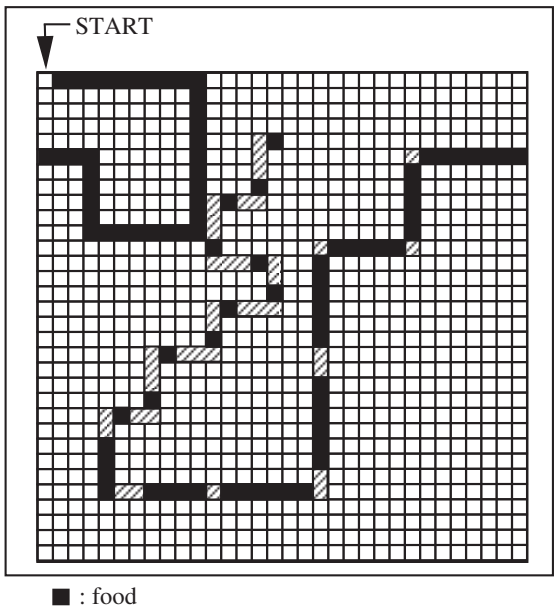



**Fig. 7.** Experiment Environment I

### 4.1 Experiment Conditions

Experiment conditions are as follows.

- Population size
  - 256 individuals
- Individuals/environment
  - 1 individual
- Selection Method
  - Roulette Model
  - Elite strategy
    - Elite size : 20

– Genetic operation
  - mutation
  - duplication
  - deletion
  - crossover( HL-type and DL-type )
  - reproduction
– I/O Facilities of Artificial Ant (see the figure 8)
  - Input pin : 5 pins
  - Output pin : 2 pins
– Fitness function
  - Fitness = Score + limit - Step + 1
    Score : units of food collected
    limit : Maximum step number allowed in this system (350 steps)
    Step : move from one square to another
    If all food is collected, AdAM is stopped.

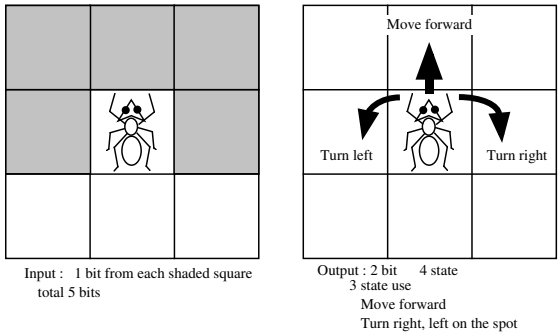



**Fig. 8.** Artificial Ant

### 4.2 Environmental Fluctuation

The effectiveness of the new AdAM was challenged by dynamically changing the food location. The food location was changed to form a model environmental fluctuation. Figure 9 indicates the changed environment.

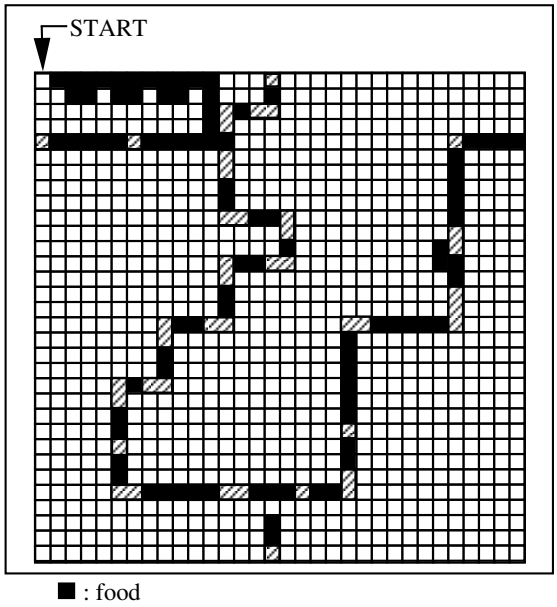



**Fig. 9.** Experiment Environment II

After individuals reaching maximal fitness appeared in environment I, the environment was changed to environment II. The maximal fitness was 293 points in environment I and 275 points in environment II.

In environment I, the best individual (artificial ant) uses only simple state transition. For the environment given, the ant does not need to judge the priority of inputs; it can collect the food quickly by moving forward.

In environment II, the ant must assess the priority for each input to collect all the food. If the ant simply moves forward, it will not be able to collect all the food within the step limit. That is, in environment II, the ant must acquire the behavior of assessing input priority in addition to those behaviors learned in environment I; environment II is more difficult and complicated than environment I.

## 5 Result

Figure 10 shows the result of the new AdAM, while figure 11 shows that of the old one; the environment was switched from I to II after individuals of maximum fitness emerged. Although the old AdAM achieved maximal fitness faster than the new one in environment I, the old AdAM spent more generations to get over 200 points than the new AdAM after the change in the environment. The new AdAM's mean of fitness values and standard deviation are higher than those of

the old AdAM. That is, individuals in the new AdAM were better in performance as a whole than those of the old AdAM. The new AdAM is superior to the old AdAM in terms of genetic diversity because its standard deviation value was bigger.

Figure 12 shows the performance of the new AdAM, while figure 13 shows that of the old AdAM in environment II. The old AdAM evolved individuals that could get high fitness faster than the new AdAM in environment II. However, the old AdAM needed more generations to develop excellent individuals than the new AdAM when the environment fluctuated.

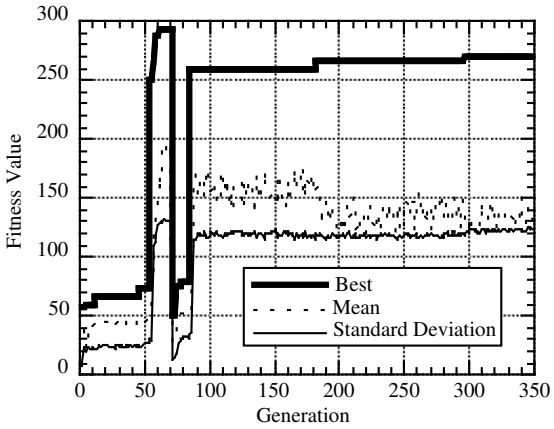



**Fig. 10.** Result of the new AdAM in fluctuating environment (I to II)

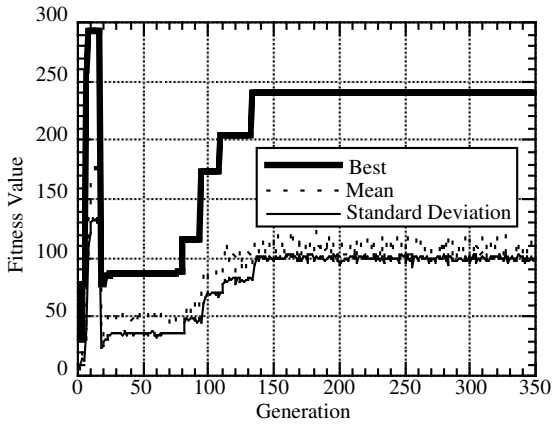



**Fig. 11.** Result of the old AdAM in fluctuating environment ( I to II)

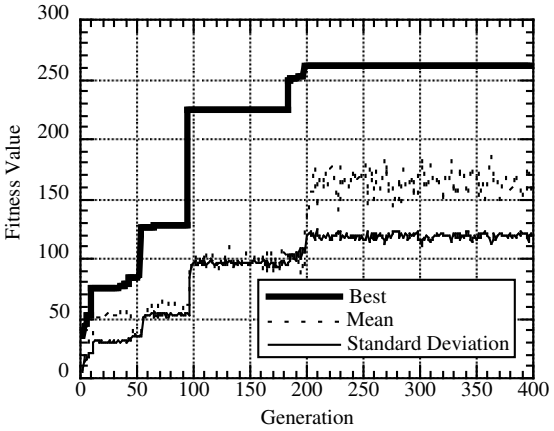



**Fig. 12.** Result of the new AdAM in environment II

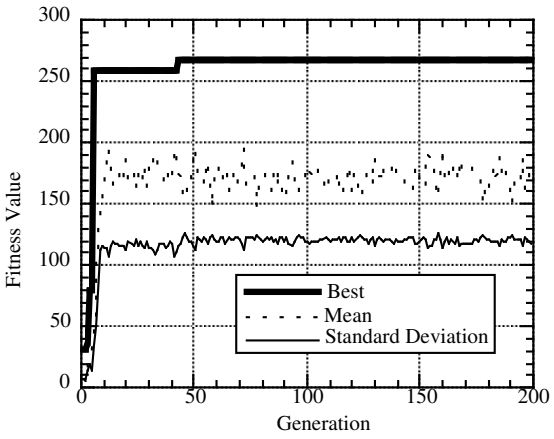



**Fig. 13.** Result of the old AdAM in environment II

## 6 Conclusion

We have proposed the new AdAM as an autonomous hardware evolution system. It offers dominant-recessive heredity to achieve genetic diversity which enhances system adaptability and robustness against changeable environments. A new chromosome model and several genetic operations were also proposed in order

to implement diploid and dominant-recessive heredity into the system. In terms of the richness of chromosome expression and the number of genetic operations possible, we can say that the new AdAM achieves genetic diversity.

We compared the new AdAM to the old one by verifying system adaptability and robustness to environment changes by simulating the behavior of an artificial ant colony. Although the old system evolved individuals that got maximal points faster than the new one, the new system offered much faster adaptation than the old one when the environment was changed. That is, individuals of the new system are robust against environment changes.

Simulation results suggest that progressive evolution, in which evolution takes place stepwise to match environment changes, is much more effective and useful as a computational framework for hardware evolution. We will continue to refine the proposed system through investigations into a new chromosome expressed by a general graph and/or progressive evolution as further studies.

## References


1. Bruce Albers et al. *MOLECULAR BIOLOGY OF THE CELL SECOND EDITION.* KYOIKUSHA, 1993.
2. Tetsuya Higuchi, Tatsuya Niwa, Toshio Tanaka, Hitoshi Iba, Hugo de Garis, and Tatsumi Furuya. Evolvable hardware – genetic based generation of electric circuitry at gate and hardware description language (HDL) levels. Technical Report 93–4, Electorotechnical Laboratory, Tsukuba, Ibaraki, Japan, 1993.
3. Hemmi Hitoshi, Jun'ichi Mizoguchi, and Katsunori Shimohara. Development and evolution of hardware behaviors. In *Artificial Life IV.* MIT Press, 1994.


This article was processed using the LaTeX macro package with LLNCS style