A Step Toward Code Granulation Space

Yinliang Zhao
Department of Computer Science, Xi'an Jiaotong University
Xi'an, 710049
zhaoy@mail.xjtu.edu.cn

Abstract

This paper proposes a code granulation space which can be applied to program construction and processing. Each granule in the space is defined as an improvement to the seed program of the space. The class tree of the seed program is partitioned into forests with context variables dependable. The boundary of granules can be specified according to both trees in the forest and shadow classes defined for improvement. Granules are organized in hierarchical style where ones at upper level are more generous than ones at low level. Granule similarity is also defined according to granule hierarchies. This Space can represent all improvements to given seed program, and can explain the fitness of programs as satisfying of granule's context within these programs. In this paper we describe details of the code granulation space, and show its effectiveness with examples.

1. Introduction

From programmer's viewpoint, a program comprises both data and code. Traditional methodologies for program construction deal with both data and code successfully in modularity, while still have fitness problems with them [1]. As an alternative to general problem solving methodologies, granular computing is emerged as a promise perspective in data organizing and processing [2, 3]. Thus this paper intends to address the fitness problems of program code using the notion of granular computing.

The primary components of granular computing are information granule, granule's boundary, hierarchical structure for universe and laws between levels of hierarchies [4, 5]. These methodologies of granular computing are efficient for applying to program data.

Code, the other half of programs, is totally different from data. Code is applied to process data in computing. Code takes such a shape that can be interpreted using its grammar. In addition, program code has semantics, control and data flow, while data do not. Therefore, partitioning code into granules is an open problem in granular computing.

In terms of software engineering methodologies, we have common partition principles with granular computing, such as divide-and-conquer, decomposition, and modularity, etc. So, it is meaningful to borrow the notion of granular computing to formalize the software engineering process in order to solve problems more efficiently. Researchers investigate software engineering methodologies from viewpoint of granular computing [7, 8].

From viewpoint of organizing and processing program code effectively in order to express its fitness, this paper proposes a code granulation space in which the fitness is satiable. The fitness of computer program [1] reflects some changes from time to time during the life cycle of programs. Those changes have an impact on the execution of the programs. More specifically, when compare programs in writing time to ones in run-time; we may find such a change possibly causes a program run in malfunction, even though we make the program perfect in writing time.

The proposed code granulation space consists of a set of granules, hierarchies of granules, and granule similarity relationship. Any program, also called individual in this paper, is represented as a hierarchy of granules. Any improvement to a program is fulfilled via a series of transitions, i.e. granule substitution, between individuals. The space is complete, so that it contains all possible improvements for a given program.

Currently, the proposed code granulation space is only applied to single inheritance object-oriented programs. The remainder of this paper is arranged as follows: Section 2 defines a partition model for program improvement with relativity to context variable; Section 3 defines granule boundary and introduces granule hierarchy construction; Section 4 is a procedure for constructing the proposed space; Section 5 is the completeness of the space; Section 6 demonstrates the space with an example; Section 7 is related work and last section is conclusion and future work.

2. Program Partitioning by Relativity to Context Variables

Programs written in object-oriented languages, OOL, in short, have class hierarchies with themselves. For programs written in single inheritance OOLs their class hierarchies can be simplified into class trees, and each program has and only has one class tree with it. We, therefore, only discuss such programs that any one of them is identical to a class tree, and do not discriminate between programs and class trees. Other situations are beyond the scope of the paper.

Let $\mathcal{P}$ be universe of single inheritance object-oriented programs. For any $T \in \mathcal{P}$, T is identical to a class tree.

Definition 2.1 (*Context variable*). For any $P \in \mathcal{P}$, a context variable is such an entity that is defined outside the address space of P, and additionally has direct effects on the execution of P.

For example, the amount of processors of a parallel machine is a context variable, which is a factor of parallel machines that affects the speedup of parallel programs which run on those machines.

Let $\mathcal{V}$ be universe of context variables. For any $P \in \mathcal{P}$, there exists a set V, $V \subseteq \mathcal{V}$, of context variables such that have direct effects on the execution of P. We write V as V_P to reflect that V is related to P.

For any $v \in V$, we may or may not capture it when writing P. We assume all $v \in V_P$ are captured for any perfect program P. When running P, any $v \in V_P$ takes values which are determined dynamically by runtime systems. However, P may still have fitness problems with it, because any $v \in V_P$ is not accessible by P itself.

For any $P \in \mathcal{P}$, and any $v \in V_P$, P is always partitioned into two parts: one has relativity to v and the other does not. Additionally, only the related part of P reflects how v affects P, and the other part is independent to v.

For example, in Amdahl's law, a program is partitioned into two parts: one is the sequential fraction of the program, the other part is parallelizable. Therefore, the speedup of the program varies only with the parallelizable part on particular parallel machines.

Programs are not beyond the impacts of context variables on themselves, no matter in writing time or in run-time. Generally, all partitions of given program P, with relativity to context variables $v_1, \ldots, v_n$, can be designated by a n-dimensional hyper-cube, which is called program-context-variable relativity hyper-cube, PVR, in short. The i-th dimension of PVR(P) indicates that P is partitioned with relativity to v_i, where the low bound and high bound of i-th dimension of PVR(P) indicate both partition parts with and without relativity to v_i, respectively.

The node k of PVR(P) indicates all possible l-order partition of P with set of context variables: $\{ v_i \mid 2^i \&\& k \mathrel{!}= 0 \}$, where l is the number of 1-bit in the binary representation of k, denoted by BITS(k) in this paper, and by l-order partition of P we obtain an ordered set of partitions with each of l context variables in some order. A simple path from node 0 to any node k in PVR(P) , where the distance of the path is less than or equal to $\lfloor \log k \rfloor$, indicating the order of l-partition with relativity to each element in $\{ v_i \mid 2^i \&\& k \mathrel{!}= 0 \}$.

Definition 2.2 (*Organelle*). For any $P \in \mathcal{P}$ and $v \in V_P$, if the partition result of P with v is a non-empty forest, denoted as FOR(P, v), then the connection v with t, $t \in$ FOR(P, v), is called a tagged class sub-tree, or an organelle.

For any $P \in \mathcal{P}$, $V \subseteq V_P$, All organelles generated from P and V is denoted as $\Pi(P, V)$ or $\Pi = (P, V)$. Given $P \in \mathcal{P}$, $\Pi(P, V_P)$ is determined, and for any $V \subseteq V' \subseteq V_P$, $\Pi(P, V) \subseteq \Pi(P, V')$.

Generally, the node k of PVR(P) has BITS(k)! partitions, where every one is a distinctive sequence of organelles generated by partitioning P with a distinctive ordered set $\{ v_i \mid 2^i \&\& k \mathrel{!}= 0 \}$.

3. Granules to Improve Programs

For $P \in \mathcal{P}$, $V_P \subseteq \mathcal{V}$, and any $\rho \in \Pi(P, V_P)$, an improvement to P is embodied by an *episome*. ρ is also precisely written as $\rho(P, t, v)$, $t \in$ FOR(P, v), $v \in V_P$.

Definition 3.1 (*Episome*). An episome is a well-formed piece of code to be added to tagged class sub-tree $\rho(P, t, v)$ for the purpose of making P fit to context variable v. An episome for organelle $\rho(P, t, v)$ is denoted as $\xi(P, t, v)$.

In this paper, we just consider such episomes that take code as a set of *shadow classes*. A shadow class has the same class name with an object class in the class tree. However, shadow classes may have their own instance variables and methods to be considered as concrete improvements to those object classes.

In OOLs, such as CLOS and Java, classes can be modified dynamically by adding them more instance variables or methods, or by deleting existed ones from them. For example, using CLOS dynamic class redefinition and method redefinition mechanism, old classes can be updated into new ones by adding, deleting slots and methods. Java override mechanism permits one

method to be substituted by another with the identical signature.

An episome, in the form of shadow classes, provides some additional methods and instance variables to object classes specified in the shadow classes. When needed, those methods or instance variables will be added to object classes dynamically in order to fulfill improvements to the program. Otherwise, these shadow classes stay invisible in the program.

Definition 3.2 (*Context*). A context $\omega(v)$ is a piece of code connected with an organelle $\rho(P, t, v)$ and an episome $\xi(P, t, v)$, and only has context variable v in its code. The context is evaluated in runtime environment in parallel with the execution of P. v maybe appear in code as a free variable. It takes values determinable by run-time systems only.

Granules are considered as improvements to programs. They have relationship to context variables, and indicate improvements with relativity to those variables. The condition that a granule is visible in program P is called *expect context* of the granule. Expect context is a piece of code generating logic result: *true* or *false*, when evaluating it in the program's environment.

Definition 3.3 (*Granule*). A granule $\sigma(P, t, v)$ is a composite of an organelle $\rho(P, t, v)$, an episome $\xi(P, t, v)$ and a context $\omega(v)$, with safe OOL semantics, as well as with its fitness to be the evaluation result of $\omega(v)$ in run-time.

Granule, generated from PVR(P), is represented as 5-tuple (*gn*, *super*, *root-class*, *context*, *shadow-classes*), where, *gn* is identity of the granule, *super* is *gn*'s parent granule, *root-class* is class identifier in P, *context* is *gn*'s expect context, and *shadow-classes* is an episome.

gn is unique to PVR(P). *super* is used to explicitly specify *gn*'s parent granule, if existed. Parent relation provides a means to organize a set of granules as a hierarchical structure in which any upper level granule is the parent of granules at its one level below. Another hierarchical relation between granules is implicit; granule $\sigma(P, t, v)$ is an ancestor of granule $\sigma(P, t', v')$ if $t' \subset t$. *context* is context code in which v is free variable and defined outside P's address space. In other words, while evaluating *context*, the value of v is needed but is determined in parallel with the execution of P, dynamically. *context* provides the granule an expect situation the episome works properly.

Figure 1 shows an example of partitions and improvements to class tree 0 with relativity of context variable u, v and w. One of 3-order partitioning results in a partition sequence with elements FOR(0, u), FOR(0, v), and FOR(0, w), in order.

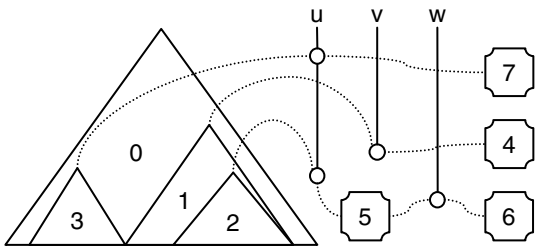

Figure 1. A example of 3-order partition of class tree 0 with relativity to context variables and 4 episomes for improving 3 sub-trees in tree 0

The result of 3-order partition of tree 0 is sequence ({tree 2, tree 3}, {tree 1}, {tree 2}). 4 episomes are shadow-class set 5, 7, 4 and 6, which are improvements to tree 2, 3, 1, and 2, respectively. Therefore, 4 granules are constructed as follows:

(g1, g0, <organelle 3>, $\omega 1(u)$, <episome 7>),

(g2, g3, <organelle 2>, $\omega 2(u)$, <episome 5>),

(g3, g0, <organelle 1>, $\omega(v)$, <episome 4>), and

(g4, g2, <organelle 2>, $\omega(w)$, <episome 6>).

g0 is a special granule, initialized with the class tree of program P and null episome, and is always at top level of every granule hierarchy constructed from P. We call g0 or class tree of P the seed and any improvement to P an individual in this paper. Every individual has one granule hierarchy and a sequence of episomes with it.

All partition results in this example are embodied in a 3-dimensional hyper-cube PVR(0, (u, v, w)). Node 0 has the seed in it. Other nodes have sets of individuals each.

4. Construction of Code Granulation Space

In this section, we formally describe a procedure of code granulation space construction.

For any $P \in \mathcal{P}$, P is a class tree denoted as $P = (C, A)$, or P(C, A), where C is a set of classes and A is arc set.

Given $V \subseteq \mathcal{V}$, suppose some elements in C are tagged with some elements in V respectively. In addition, for any $c \in C$ and any $v \in V$, at most one node on the path from root(P) to node c is tagged with v.

For any $c \in C$, there exists $v \in V$, if c has tag v then we call the sub-tree with root c in P a tagged class sub-tree, which is denoted as T(P, c, v).

Given P and V, the universe set of tagged sub-trees by above tagging process is called a partition of P with V, and denoted as $\Pi = (P, V)$.

Definition 4.1. An ordered partition $\boldsymbol{\Pi} = (P, \boldsymbol{V})$ is an ordered tree, in which any node is a tagged sub-tree of P,

such that for each arc from node T(P, $c1$, $v1$) $\in$ $\mathbf{\Pi}$ to node T(P, $c2$, $v2$) $\in$ $\mathbf{\Pi}$,

$c1 \prec c2 \rightarrow$ T(P, $c1$, $v1$) $\prec$ T(P, $c2$, $v2$), or

$c1 = c2 \rightarrow$ (T(P, $c1$, $v1$) $\prec$ T(P, $c2$, $v2$) $\rightleftarrows$ $v1 \prec v2$).

By topological sorting (Π,$\prec$), we get $\mathbf{\Pi}$.

4.1. Construction of Granules and Hierarchies

By connecting a tagged class sub-tree T(P, c, v) with both a set of shadow classes and a piece of context code $\omega(v)$, we get a granule.

Components of any granule are accessed using the following accessors: tree(g) is g's class sub-tree; ctx-var(g) is v; context(g) is $\omega(v)$; shadow-classes(g) is a list of g's shadow-classes.

Seed granule is a special granule, where, let $g0$ be seed granule, tree($g0$) is P and both context($g0$) and shadow-classes($g0$) are $\perp$. Seed granule is unique for any Π = (P, V), $P \in \mathcal{P}$ and $V \subseteq \mathcal{V}$. In this paper, we do not differentiate between the seed granule and P, and both are called seed, in short.

For each T(P, c, v) in $\mathbf{\Pi}$, we construct a granule. Then we get an ordered granule hierarchy **G** such that, for any $g1$, $g2 \in$ **G**, tree($g1$) $\prec$ tree($g2$) $\rightarrow$ $g1 \prec g2$.

Individual **H**(P, **V**) is an ordered granule hierarchy which has the seed granule to be added to **G** as the parent of every top level granule in **G**.

Population $\mathcal{K}(P, V)$ is the universal set of individuals derived from a specified program P and set of context variables V. For any $W \subseteq V$, and G is derived from Π = (P, W), then, for any partial order $\prec$ on W, **G** = (G, $\prec$) $\in$ $\mathcal{K}(P, V)$.

The structure of $\mathcal{K}(P, V)$ is a $^{\#}V$-dimensional hyper-cube. Not to lose the general, V is linearized as **V** = (v_0, …, v_{n-1}), in which $n = {}^{\#}V$, index(v_i, **V**) = i, and **V**[i] = v_i. Therefore, the structure of $\mathcal{K}(P, V)$ is described by the following procedure:

{j=0; for (i=0; i<n; i++)if (2^i&&k) **W**[j++]=**V**[i]; }.

Then, the length of **W** is j-1. **W** takes one of (j-1)! different orders.

For all **W** with same set of context variables, $\mathbf{\Pi}$ = (P, **W**) is certain and derives **G**, then we can conclude that all individuals **H**(P, **W**) are located at node k, and **W** is corresponding to one and only one of the shortest path from node 0 to node k in $\mathcal{K}(P, V)$.

4.2. Granule Similarity Relationship

For any $g1$, $g2 \in$ **H**, $g1$ is said corresponding to $g2$ if tree($g1$) = tree($g2$) and ctx-var($g1$) = ctx-var($g2$). If $g1$ is corresponding to $g2$ then $g2$ is corresponding to $g1$.

For any **H'**, **D'** $\in \mathcal{K}(P, V)$, **H'** is corresponding to **D'** if the following procedure is satisfied:

Step 1: Let **H** be the hierarchy derived from **H'** by removing some sub-trees from it. Let **D** be derived from **D'** in same way. Then, the length of **H** is equal to the length of **D** and,

Step 2: for each i, $0 \le i <$ length(**H**), **H**[i] and **D**[i] are two corresponding granules; for any $g \in$ **H**, if (**H**[i], g) is an arc in **H** then (**D**[i], **D**[index(g, **H**)]) is an arc in **D**; for any $g \in$ **D**, if pair (**D**[i], g) is an arc in **D** then pair (**H**[i], **H**[index(g, **D**)]) is an arc in **H**.

For any granule $g1$ in **H** and $g2$ in **D**, $g1$ and $g2$ have similarity relationship with each other, if the pair $g1$ and $g2$ is in the result of the following recursive procedure:

Step 1: if $g1$ = root(**H**) and $g2$ = root(**D**) are identical, then they are similar granules to each other.

Step 2: if the children($g1$, **H**) and children($g2$, **D**) are in equal length (>0) and their elements are corresponding one by one, then elements in every pair are similar granules. Otherwise it returns to one upper recursion level.

Step 3: Recursively repeat *step 2* with every pair from children($g1$, **H**) and children($g2$, **D**).

5. Completeness

Theorem 1: Any improvement of program P with set of context variables V is an element of space $\mathcal{K}(P, V)$.

Proof: Assume that **Q** is an improvement of P, then, without lost in general, suppose that **Q** is derived from **G**. Thus, according to individual construction in Section 4, we derive $\mathbf{\Pi}$(P, **V**) from **G** by the following procedure:

(1) Let **B** is a copy of **G**;

(2) Set **B**[i] with tree(**G**[i]), for each $1 \le i <$ length(**G**);

(3) Remove root(**G**) from **B**;

(4) **B** is $\mathbf{\Pi}$(P, **V**).

$\mathbf{\Pi}$(P, **V**) is an ordered partition of P with **V** as defined in definition 4.1. □

Theorem 2: For any **Q** $\in \mathcal{K}$, $g \in$ **Q**, if **R** is derived from **Q** by substituting g with its similar granule, then **R** $\in \mathcal{K}$.

Proof: Let $\mathbf{\Pi}$(P, **V**) be an ordered partition from which **Q** is generated through granule construction. Let v

be ctx-var(g), and $v \in \mathbf{V}$. Without lost in general, suppose $\mathbf{V}[i] = v$, $1 \le i <$ length($\mathbf{V}$). Let $g1$ be a similar granule to g, and $\mathbf{\Pi'}$ the ordered partition whose sub-tree tree(g) is substituted by sub-tree tree($g1$).

Let c = root(tree(g)), c' be parent of c, and g' parent of g, then ctx-var(g') $\prec v$. $g1$ and g are similar to each other, so tree($g1$) = tree(g).

We only prove that $\mathbf{\Pi'}$ satisfies the definition 4.1 as following.

(1) if c = root(tree(g')) then ctx-var(g') $\prec v$ and tree(g') $\prec$ tree(g) = tree($g1$).

(2) if root(tree(g')) $\prec c$ then tree(g') $\prec$ tree(g) = tree($g1$).

(3) By granule substitution, we know $\mathbf{\Pi'}$ is corresponding to $\mathbf{\Pi}$. □

6. An Example

This section gives code granulation space $\mathcal{K}(F, V)$, where F is a program to compute factorial number for given input number, and V = {c t s} is set of context variables as show in the following:

- c specifies the control style of computation and takes possible value: i, iterative; or r, recursive;
- t specifies the type size of data type short, int and bignum, respectively;
- s specifies the stack size of the recursive computation.

Program F contains one default class S and 4 classes described as the following piece of code:

```
class R {
  int n;
}
class E extends R {
  output(int f){…}
}
class D extends R {
  input(){…/*write to this.n*/}
}
class C extends D {
  int fact(){return(-1);}
}
```

F, an ordered class tree, has S as its root. S has a child R which has two children E and D. D has a child C. In the *main* method, an object k of class C and e of class E are created. The top level loop of *main* method looks like:

```
{k.input(); e.output( k.fact() ); }.
```

We assume the default behavior of F is just print -1. A list representation of F is (S (R (E) (D (C)))). We have:

$\mathbf{\Pi}(F, (c)) = ((T(F, C, c)))$;

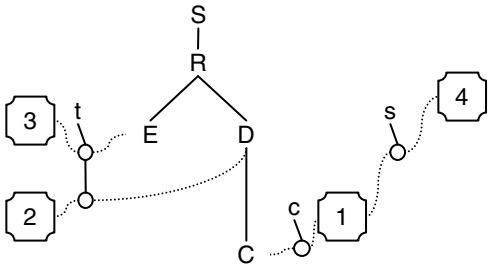

Figure 2. $\mathbf{\Pi}(F, (t\ c\ s)) = \mathbf{\Pi}(F, (c\ t\ s)) = \mathbf{\Pi}(F, (c\ s\ t))$ = ((T(F, E, t))(T(F, D, t) ((T(F, C, c) (T(F, C, s))))))

$\mathbf{\Pi}(F, (t)) = ((T(F, E, t)) (T(F, D, t)))$;

$\mathbf{\Pi}(F, (c\ t)) = \mathbf{\Pi}(F, (t\ c)) = ((T(F, E, t)) (T(F, D, t) (T(F, C, c))))$;

$\mathbf{\Pi}(F, (c\ s)) = ((T(F, C, c) (T(F, C, s)))$;

$\mathbf{\Pi}(F, (s\ c)) = ((T(F, C, s) (T(F, C, c)))$.

The partition of F in the example is shown in Figure 2.

G is constructed by adding context code and shadow classes to every tagged tree in $\mathbf{\Pi}$. **G** = (($g1$) ($g2$ ($g3$ ($g4$)))), where, tree($g1$) = T(F, E, t), tree($g2$) = T(F, D, t), tree($g3$) = (T(F, C, c), tree($g4$) = T(F, C, s). To add the special granule $g0$ to **G**, where tree($g0$) = F, context($g0$) = $\perp$ and shadow-class set is $\perp$, we get individual **H** = ($g0$ ($g1$) ($g2$ ($g3$ (g4)))). **H** is located in node 7 of $\mathcal{K}(F, V)$.

Table 1. Possible Granules.

Tree	Context	Shadow classes
(C)	c==i	{Ci}
(C)	c==r	{Cr}
(E)	t[short]==2	{Es}
(D(C))	t[short]==2∧n<=7	{Ds, Cis}, {Ds, Crs}
(E)	t[int]==4	{Ei}
(D(C))	t[int]==4∧n<=12	{Di, Cri}, {Di, Cii}
(E)	t[bignum]	{Eb}
(D(C))	t[bignum]	{Db, Cib}, {Db, Crb}
(C)	s>=16000∧n<=184	{Crb}

Some possible granules of this example are shown in Table 1. Typical shadow classes in this table can be coded as follows:

```
shadow-class Cr {
  int fact(){return fact1(this.n);}
  int fact1(int n){…/*recursive n!*/}
}
shadow-class Ci {
  int fact(){…/*iterative n!*/} }
```

```
shadow-class Cib {
  bignum fact(){…/*iterative n!*/}
}
shadow-class Crb {
  bignum fact(){return fact1(this.n);}
  bignum fact1(bignum n){…/*rec. n!*/}
}
shadow-class Db {
  bignum n;
  input(){…/*write to this.n*/}
}
```

7. Related Work

Yao and Zhong [6] propose a methodology to partition granules by similarity of attribute values, which can be applied in data processing.

Han [7] describes program partition for logic program analysis, which is based on relations and subprograms. This work intends to optimize the query mechanism in logic programming.

Qin and He [8, 9] propose a program partition approach for hardware and software co-design, which consists of analysis and parts. The program analysis phase provides operation moving between software and hardware parts. The composition part is a collection of syntax-based splitting rules.

Zhu [10] proposes a method of granulation by rolification, i.e., roles and agents, for improving design process of software engineering. Han and Dong [11] investigate granular computing from the perspectives of all software engineering processes. Wu [12] studies existing programming language features in granular computing viewpoint.

8. Conclusions and Future Work

This paper proposes a code granulation space for program construction, which consists of a set of granules, granule hierarchies and granule similarity relationship. It has the following conclusions:

With the relativity to context variables, class trees can be partitioned into two parts for each context variable: one is related to the context variable and the other does not. The related part is a forest of class sub-trees. The class trees can also be partitioned in order for the sequence of context variables.

Program improvement is embodied to attaching shadow classes to class sub-trees of the program. Granule is used to specify such improvement in a context. Granule hierarchy, derived from set of granules and partition order, specifies improved program, i.e., individual.

All granule hierarchies derived from one class tree are comparative with each other. Similar granules locate between different comparative hierarchies.

The proposed code granulation space is complete. Every improvement to a program belongs to the space of the program. By granule substitution, one individual can evolve into another individual in the same space.

The future work is to extend the space generally. Formal description and verification of semantics safety for code granulation space is fundamental to program code organizing and processing through the granular computing approach. Granulation spaces for class hierarchies (not a tree) and other program structures are useful but not yet addressed. The way to making granules can also be extended using method combination.

9. References

[1] Zhao, Yinliang: Granule-Oriented Programming. ACM SIGPLAN NOTICES 39(12), 107-118, 2004.

[2] Bargiela, A. and Pedrycz, W.: The Roots of Granular Computing. In: Proc. of the IEEE Int'l Conf. on Granular Computing. Atlanta, USA (2006) 806-809.

[3] Lin, T.Y.: Granular Computing: A Problem Solving Paradigm. In: Proc. of the IEEE Int'l Conf. on Fuzzy Systems. Reno, Nevada, USA (2005) 132-137.

[4] Yao, Y.Y.: The Art of Granular Computing. In: Proc. of the Int'l Conf. on Rough Set and Emerging Intelligent Systems Paradigms, LNAI 4585, 2007, pp. 101-112.

[5] Pedrycz, W.: Granular computing: an introduction. In: Proc. of the IFSA World Congress and 20th NAFIPS Int'l Conf.. Vancouver, BC, Canada. Vol. 3 (2001) 1349-1354.

[6] Yao, Y.Y. and Zhong, N.: Granular Computing Using Information Tables. In: Lin, T.Y., et.al. (Eds.), Data Mining, Rough Sets and Granular Computing, Physica-Verlag, Heidelberg, (2002) 102-124.

[7] Han, J.L.: Program Partition and Logic Program Analysis. IEEE Trans. Software Eng. 21(12): 959-968 (1995)

[8] Qin, S.C., He, Jifeng, et al.: An Algebraic Hardware / Software Partitioning Algorithm. J. Comput. Sci. Technol. 17(3): 284-294 (2002)

[9] Qin, S.C. and He, Jifeng: Partitioning Program into Hardware and Software. In: Proc. of Asia-Pacific Software Engineering Conference. Macau, China (2001) 309-316

[10] Zhu, Haibin: Granular Problem Solving and Software Engineering. In: Proc. of the IEEE Int'l Conf. on Granular Computing. Hangzhou, China, (2008) 859-864

[11] Han, J. and Dong, J.: Perspectives of Granular Computing in Software Engineering. In: Proc. of the IEEE Int'l Conf. on Granular Computing. Silicon Valley, USA, (2007) 66-71.

[12] Wu, T.: Granular Computing in Programming Language Design. In: Proc. of the IEEE Int'l Conf. on Granular Computing. Beijing, China, (2005) 296-302.