Granule-Oriented Programming

Yinliang Zhao

Technical Report CS-2004-001

March, 2004

Copyright ©2004, Yinliang Zhao

Department of Computer Science and Technology
Xi'an Jiaotong University
Xi'an 710049, P.R.CHINA

Department of Computer Science
University of Regina
Regina, Saskatchewan, CANADA S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0467-3

Granule-Oriented Programming

Yinliang Zhao[†] [‡]
[†] Department of Computer Science and Technology
Xi'an Jiaotong University, Xi'an 710049, P.R.China
zhaoy@mail.xjtu.edu.cn

[‡] Department of Computer Science
University of Regina, Regina, Canada S4S 0A2

ABSTRACT
A program will become obsolete or lower effectiveness in solving domain problems due to many reasons. One main reason is because the program becomes unfitting to its context, which is defined as a sum of functionalities of all what support the program solving the domain problems, for instance, a runtime environment, meta-strategies in the domain, etc. This unfitting phenomenon exists in many complex systems, causing them premature end of their life cycle or a decrease in performance or accuracy in solving problems. In existing programming systems we pay little attention to unfitness of a program to its context, in language expressivity. Granule-oriented programming is an evolvement metaphor in which programs are "ground" into code ingredients in order to localize unfitting parts of a program as explicitly as possible, and then "compound" them into granular output code in which the code granulation space is formed. Code granulation space is an expression of a program in multiple-abstraction framework. The goal of building code granulation space for a program is to localize unfitness in a well-formed and multi-layered framework. In this paper, we propose and briefly describe the notion of granule-oriented programming.

Categories and Subject Descriptors
D.3.3 [**Programming languages**]: Language Constructs and Features –*Control structures, frameworks*.

General Terms
Languages, Design.

Keywords
Reflection, Aspect-Oriented-Programming, Granule-oriented programming, context distribution, Code granulation space, Object-oriented programming.

1. INTRODUCTION
In this paper, we present an overview of our research on programming language expressivity. The goal of this work is to make it possible for programs to capture all of the important expression issues of the system's structure and behavior, including not only its functionality, but also whatever supports its solving domain problems such as exception handling support, runtime support, coordination support with other programs, etc.

Our current work is based on the belief that programming languages based on single- or multiple-abstraction framework – procedures, constraints, aspects, whatever – are ultimately inadequate for many complex systems. The reason is that a complex system must be programmed into live fitting to its context such as its domain, its exception, its execution, its coordination with others, which is outside of almost existing programming language's expressivity.

"Unfitting" phenomenon often occurs in real complex software systems. It mostly happens when a program is forced to solve fresh problems in the domain. In other words, the program focuses on different problems in its domain from ones that it is programmed to solve before. This may cause the existing programs in the domain not being able to solve the new domain problems properly. For this situation, the system will become obsolete, that is, it has reached to the end of its life cycle, and new program will be applied to solve current domain problems. Another moment unfitting phenomenon happens is when the program faces new runtime support systems such as improved memory management, communication means, or even new machine, etc. So invariable agreement between program and its context (focuses on

the domain, or different runtime support, etc.) will make premature end of the program's life cycle, or an improper (in low efficiency, inaccuracy) continuous execution.

One of the explanations for which unfitting phenomenon happens, lays in the limitation of existing programming language's expressivity. So we should add some facilities to programming languages for explicit expression of localizing unfitting by programming on context. In other words, the program should be able to access to, interact with its context to deal with unfitness.

This conclusion has led us to develop a concept we call Granule-Oriented Programming (GOP). GOP is an evolvement metaphor in which programs are "ground" into code ingredients in order to locate unfitting parts of a program as explicitly as possible. These code ingredients are compounded into program components, called code granules. Code granules are organized as a granulation space in which we can control unfitness from multi-level abstraction such as zooming-in or zooming-out. In GOP, we pay attention to code granules, for instance, their evolution from one program to another. Granules can be layered through one or more lower level granules being compounded into high-level granules. We believe that programming on the zooming-in and zooming-out along with granulation layers in the granulation space is important to localize unfitting.

From point view of GOP, a domain is described using one or more domain problem solving cases. And the result of GOP is to produce granular output code from these cases. In this paper, we propose an overview of granule-oriented programming. In Section 2, from the observation of unfitting phenomenon, the basic elements of GOP are studied. Two GOP examples are described in Section 3 and 4, respectively. In Section 5 some important issues are discussed. Related work is described in Section 6, and conclusion and future work are given in last section.

2. UNFITTING

The basic limitation of programming language is that the program's context is not easily programmed on as domain problem solving does. In other words, the program's context can not been seen in programming. Here the term, program's context, is employed to indicate all what are related to which the program is being processed (programming phrase) and running (execution phrase). The context is defined as a sum of functionalities of all what support the program solving the domain problems. In classical programming, programmers are forced to use language's facilities to express how to solve domain problems in the language's abstraction level. There is an invariable, no doubtful and static agreement between the program and its context. In other words, programmers may say that it's the other systems' responsibility when they face the above difficulties. For instance, to logic programmers, keeping memory reference locality in logic programming should be the compiler's work or the operating systems' responsibility. But in reality, it is hard to control the memory reference locality of knowledge representation, such as frame[13], in logic programming outside the program, although some garbage collection algorithms tend to localize object representation in memory.

The agreement between a program and its context will be broken down in many situations, for instance, in the moment that the domain has changed, or systems that are supporting execution of the program have become new, etc. A classic example about the agreement between a program and its context is remote object invocation. For example, there is an agreement between remote object and client program to keep the consistence of name binding and parameter passing[12]. Assume the remote object is modified for some reason, for instance, an extra parameter is added to the interface of a method of the object; therefore, the agreement is broken down, causing an exception to the client program in most cases. In this example, the remote object invocation is to be viewed as the client program's context. It is lack of such facilities in the language that the program's context (coordination with the remote object) is being programmed during life cycle of the program. The program will become unfitting to its context at the moment the agreement is broken down.

Many linguistic mechanisms have been developed to deal with special cases of this problem (i.e. add/remove method, reflective facilities, reuse, maintenance), but a great deal of the complexity in real world code still appears to come from cases where the language fails to provide adequate support for a secondary, but still important, programming facilities for managing unfitting phenomenon.

Some cases of this problem can be dealt with, in limited extent, by program online upgrade mechanism[1]. In above example, the agreement is extended with remote object modification concerns. Then client program can be designed as a polymorphism style for adapting to the notification of remote object update.

But there are still some cases where programming on the program's context is beyond program's ability to process due to lack of adequate means in the language expressivity. One good example happens in distributed computing. There has been a local data process system which using remote data access to get data from some data centers. Along with performance variability of the network and servers the system is using, data process algorithms easily become unfitting to data access. This is because the data process strategies have to match with data access rate for the lowest overhead theoretically, but in reality, it is hard to know either how data access rate varies, or all alternatives for balancing between data process and data access as much as possible.

A natural thinking of a solution to unfitting problem lays in that localizing unfitting code ingredients and provide alternatives for them in programming. Formally, suppose we have a program P0 fitting to context T0, as well as a context transition for P0 from T0 to T1. P0 is then unfitting to T1. So we find another program which either fits to T1 or fits to $T1 \setminus T0$. One solution is either using P1 in place of P0 in context T1, or compounding P0 and P1 where P0 for context $T0 \cap T1$ and P1 for context $T1 \setminus T0$ (the compound of P0 and P1 is viewed as the program we want for the context T1).

Moreover, we treat the program P0 and P1 as set of granules G1 and G2, respectively. Any granule in G1 or G2 faces a similar situation as program P0 and P1 do.

Table 1. Some code granules and contexts

Domain	**Example of Code Granule**	**Part of Context**
parallel computing	task tasks mapped to a processor	scheduler, communication, synchronization, processor
concurrency	thread	scheduler, synchronization
logic programming	goal derivation, clause tree search, variable substitution	representation, language, environment, continuation
object-oriented programming	object, class, method, method-combination	representation, sub-classing, message-passing, runtime-support
distributed systems	resources	remote object invocation, protocol

We believe this unfitting phenomenon hooked up the complexity of most existing complex software systems. Granule's fitting to its context is at the heart of much of the complexity in those software systems. The goal of Granule-Oriented Programming is to make it possible to deal with unfitness of a system as an operational way. We want to allow programmers first capture primary problem solving cases in the domain, express each of them as code granules, and then compound them into output code.

Examples of the kinds of granules we believe GOP should allow programmers to think and program in terms of are shown in Table 1.

2.1 A Methodology

We propose a methodology of granule-oriented programming as shown in Figure 1. Here we use the methodology, reasoning by examples to deal with unfitness to domain. The concept primary problem solving (PPS) is employed to express an incremental effort in programming. In other words, programmers capture a context snapshot for a domain, and write out a program that has an invariable agreement with the determined context. Programmers also derive

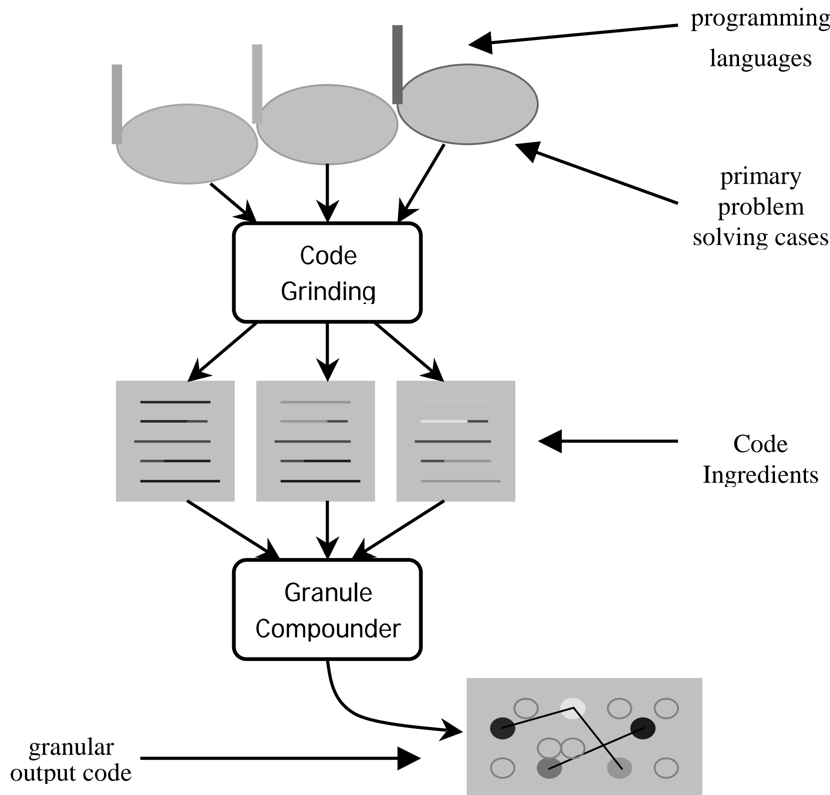

Figure 1. The basic elements of a granule-oriented programming system

programs from some primary problem solving cases of the domain. We suppose that a PPS is expressed in separate programs, which maybe written using some separate languages with each other.

By Grinding, programmers find code ingredients from each PPS and make similarity relationship between them.

An ingredient is easily to be considered as part of code, such as function, procedure, class, method, etc. The goal of grinding is to locate unfitting things by code similarity analysis. We hope similar code ingredient to be compounded into granules. Granules comprise of base granules, which are directly derived from code ingredients, as well as high-level granules.

All granules from each PPS's granules are compounded into a new PPS based on predefined granulation principles such as *add class/method*.

A similar situation occurs in parallel programming. The problem can be decomposed into a group of tasks using Ian Foster's PCAM (partition, communication, aggregation, and mapping) methodology[5], the produced tasks can be viewed as granules with coherence of concurrency. Mapping can be viewed as a kind of compound by which each high-level granule is a group of granules that are mapped to the same processor unit in the parallel system. The goal of such granulation of the program is to gain load balance.

3. AN EXAMPLE OF GRANULE-ORIENTED PROGRAMMING

As a first example of granule-oriented programming, we present a granular output code for three cases of primary problem solving in a simple and classical problem domain, sorting using *quicksort*. As a GOP example, this sorting domain is just used for demonstrating how unfitting phenomenon occurs and the significance the granules have been formed. Another reason we adopt GOP in sorting domain's problem solving is that these typical and widely accepted sorting algorithms can be programmed in an innovative way.

Though the problem domain is simple, the matching combination of sorting algorithms and their contexts is large. As an example, we just choose three primary problem-solving (PPS) cases as follows.

PPS0. Data are read from an input file and sorted in main memory using quicksort. The results are written to an output file.

PPS1. Data are read from a serial port (or a stream) and sorted in main memory using quicksort. The results are written to an output file. In this problem, we suppose the data access is significantly slow, so it may be better to sort currently arrived data each time and read data continuously in a separate thread. At last, merge all sorted results to produce the final results.

PPS2. Data are read from an input file and sorted in bounded physical memory (no virtual memory is allowed). Partition of that data file maybe needed because of memory size. As a result, external merge of quicksort-ed results is needed and by it the final results are produced.

The differences between contexts corresponding to these PPS's are that the context of PPS1 indicates data access restricted, and the context of PPS'2 indicates memory size restricted. The program of any PPS may be unfitting to the context of another PPS as shown in Table 2.

Table 2. Fitness of PPS's to contexts

	context0	**context1**	**context2**
PPS0	fitting	data source, performance	fitting (special case) unfitting
PPS1	data source, performance	fitting	data source, fitting(special case) unfitting
PPS2	fitting	data source, performance	fitting

Every PPS is assumed that is fitting to its own context in the table, for example, PPS0 is fitting to context0. PPS2 is also fitting to context0. PPS0 is unfitting to context1 because the program of PPS0 reads data from file instead of a port as context1 indicates. Another unfitting is that PPS0 will slow down in the context1 when the data access rate decrease to some extent. For the context2, PPS0 will be ok when the memory is happen to contain the input data, which is a special case, otherwise the program will crash down in the context. Similar situations occur in the other PPS – context combinations as shown Table 2.

From the observation to unfitting phenomenon in this example, we can conclude that a program becomes obsolete (cannot be used anymore) or in low performance if it is unfitting to its current context. But its code ingredients can be reused in another program. One of the granulation space compounded from the code ingredients of each PPS is shown as Figure 2.

Granules are compounded from code ingredients of each PPS, or from other low level granules. For instance, granule <quicksort: arr, i, j> is compounded from code ingredients which appears in every PPS:

```
void quicksort(int arr[],int i,int j){
  int k;
  if (i<j) {
    k = partition(arr, i, j);
    quicksort(arr, i, k);
    quicksort(arr, k+1, j);
  }
}
```

For this granule, every code ingredient: `quichsort` in PPS's is fitting to the context <`arr, i, j` >. We use the form <*g*:*c*> to represent a granule, where *g* is the granule's name and *c* is its context. Sometimes we ignore *c* when we

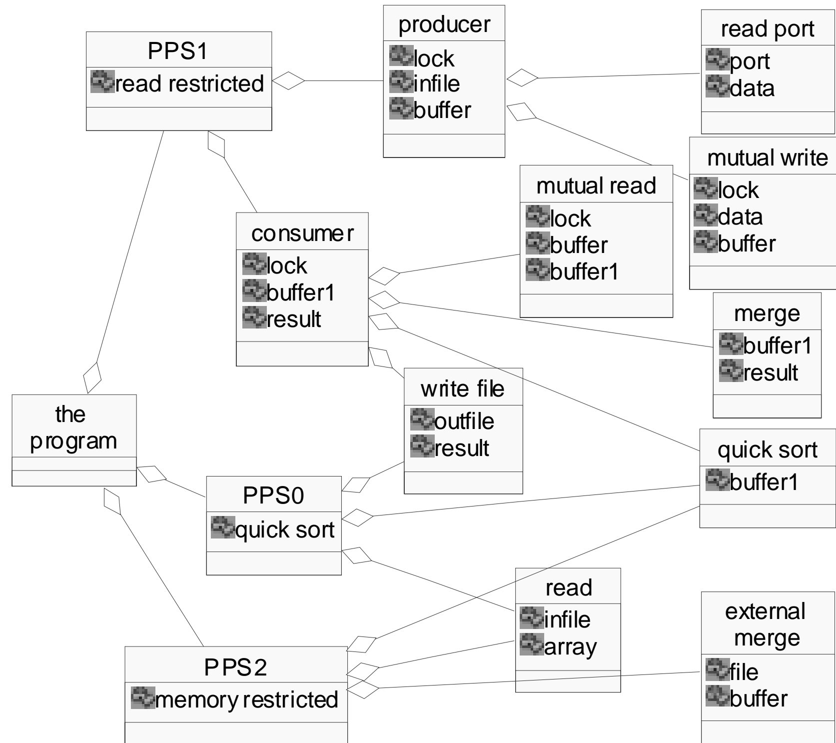

Figure 2. A granulation space of quicksort example

are not interested in. Granule <`producer:`> is constructed by compounding granule <`read_port:`> and <`mutual_write:`> as shown in Figure 2, which is borrowed from UML class diagram where class name indicates granule name and attributes to context elements. Those low-level granules can be aggregated in their parent granule through separate code (compounding) as follows:

```
#define BSIZE 64
#define ASIZE 128000
struct {int *buf;int next;int notify;} buffer;
struct {int *buf;int len;} buffer1, area;
pthread_mutex_t lock;
void merge(){
  int i,j,k;
  j=area.len-1;
  if(j<0){
    for(i=0;i<buffer1.len;i++)
      area.buf[i]=buffer1.buf[i];
  }else{
    i=buffer1.len-1;
    k=j+i+1;
    while(i>=0){
      if(area.buf[j]<buffer1.buf[i])area.buf[k--]=buffer1.buf[i--];
      else area.buf[k--]=area.buf[j--];
    }
```

```
  }
  area.len+=buffer1.len;
}
int read_port(int *data, FILE *in){
  /*Delay simulation*/
  return(fscanf(in,"%d ", data)==EOF);
}
void mutual_write(int data){
  int flag=0;
  while (!flag){
    pthread_mutex_lock(&lock);
    if (buffer.next<BSIZE){
      buffer.buf[buffer.next++]=data;
      flag=1;
    }
    pthread_mutex_unlock(&lock);
  }
}
void exception(char *p){
  printf("ERROR: %s.\n", p);
  exit(0);
}
void write_file(FILE *out, int len, int *array){
  int i;
  for(i=0;i<len;i++)fprintf(out,"%d ",array[i]);
  fclose(out);
}
int mutual_read(){
  int flag=0, rsl;
  while(!flag){
    pthread_mutex_lock(&lock);
    if (buffer.next>0){
      memcpy(buffer1.buf, buffer.buf, sizeof(int)*buffer.next);
      buffer1.len=buffer.next;
      buffer.next=0;
      flag=1; rsl=0;
    }else if (buffer.notify>0)flag=rsl=1;
    pthread_mutex_unlock(&lock);
  }
  return(rsl);
}
void notify(){
    pthread_mutex_lock(&lock);
    buffer.notify=1;
    pthread_mutex_unlock(&lock);
}
void producer(FILE *in){
  int tmp;
  while(!(read_port(&tmp, in)))mutual_write(tmp);
  notify();
}
void consumer(void ){
  while(!mutual_read()){
    quicksort(buffer1.buf,0,buffer1.len-1);
    if(area.len+buffer1.len>ASIZE) exception("Area overflow");
    merge();
  }
```

```
#define MEMORY_SIZE 1024
void external_Merge(int array[],int len,char *fn){
  FILE *fp,*fpnew;
  int index=0,value;
  if ((fpnew=fopen(".tmpfile","w"))==NULL)exception("File open");
  if ((fp=fopen(fn,"r"))==NULL)exception("File open");
  while (fscanf(fp,"%d ",&value)!=EOF){
    while(index<len)
      if(array[index]<value)fprintf(fpnew,"%d ",array[index++]);
      else break;
    fprintf(fpnew,"%d ",value);
  }
  while(index<len)fprintf(fpnew,"%d ",array[index++]);
  fclose(fpnew);
  remove(fn);
  rename(".tmpfile",fn);
}
int main(int argc,char **argv){
  int len,array[MEMORY_SIZE];
  FILE *in;
  char outfile[20];
  if(argc!=3)exception("Parameters");
  if((in=fopen(argv[1],"r"))==NULL)exception("File open");
  strcpy(outfile,argv[2]);
  if(!creat(outfile,"r"))exception("File create");
  while(len=read(in, array)){
      quicksort(array,0,len-1);
      external_Merge(array,len,outfile);
    }
  fclose(in);
}
```

PPS2

{BSIZE/MEMORY_SIZE}

```
#define BSIZE 20480
int partition(int arr[],int left, int right){
/*ignored*/}
void quicksort(int arr[],int i,int j){
/*ignored*/}
void exception(char *p){
/*simplified exception handler*/
  printf("ERROR: %s.\n", p);
  exit(0);
}
int read(FILE *in, int array[]){
  int i=0;
  while (!feof(in)&&(i<BSIZE)){
    fscanf(in,"%d ",&array[i++]);
  }
  return i;
}
void write_file(FILE *out, int len, int *array){
  int i;
 for(i=0;i<len;i++)fprintf(out,"%d ",array[i]);
  fclose(out);
}
```

PPS0

With full similarity

```
}
main(int argc, char **argv){
  pthread_t p;
  FILE *in,*out;
  if(argc!=3)exception("Parameters");
  if((in=fopen(argv[1],"r"))==NULL)exception("File open r")
  if ((out=fopen(argv[2],"w"))==NULL)exception(File open w);
  pthread_mutex_init (&lock,NULL);
  buffer.buf=malloc(BSIZE*sizeof(int));
  buffer1.buf=malloc(BSIZE*sizeof(int));
  area.buf=malloc(ASIZE*sizeof(int));
  if(pthread_create(&p, NULL, (void *)producer, in))exception("Thread create");
  consumer();
  fclose(in);
  write_file(out,area.len, area.buf);
}
```

<exception:> is an simplified exception handler used in all three PPS's. <write_file:> both in PPS0 and PPS1 have all similarities between them. <read:> in PPS0 is similar with one in PPS2 under a substitution {BSIZE/MEMORY_SIZE}. PPS1 is fully listed above in which we can analyze how high-level granules are compounded from low-level granules (are indicated by a underline). In the listing of PPS0, the main program is ignored. In the listing of PP2, if we move in the five granules as dashed arrows indicate, we will get a fully listing of PPS2.

The final granular output code is a new program, whose context is $context0 \vee context1 \vee context2 \vee (context1 \wedge context2)$.

4. A SECOND EXAMPLE OF GRANULE-ORIENTED PROGRAMMING

This is a more complex example than sorting, called Molog[18]. We just pay attention to code ingredients with similarities. Similar code ingredients from different PPS are combined into code granules for fitness.

The problem domain of this example is logic programming, i.e. Prolog. Suppose we have the following two PPS's. The first PPS, PPS0, is given as a Prolog programming system. For simplicity we just consider it as a Prolog interpreter. This program is written using Lisp language[16]. The second PPS, PPS1, is a meta-reasoning in logic programming[2][3], also called Reflective Prolog.

The base level in the meta-reasoning consists of rules and facts, while the metalevel consists of meta-rules and meta-facts. Base level goals are satisfied using base level clauses. For a goal the first time is failed at base level, a reflect-up occurs in which a corresponding metalevel goal is constructed using naming mechanism. If this goal is failed then the corresponding base level goal is failed, otherwise the metalevel reasoning is continuing according to the meta-evaluation rules. By the naming mechanism, entities are referenced as their names, called meta-constant which is accessible at metalevel. Meta-constants are de-referenced as their connected base level entities. We want to implement such mechanisms of Reflective Prolog, PPS1, in the program of PPS0. These mechanisms also can be implemented using fully implemented Prolog system.

Here is an example of Reflective Prolog's mechanisms. For fully understand of the mechanisms please refer to [2][3].

```
/*base level*/
p(a).
/*metalevel*/
equivalent("b","a").
/*meta-evaluation rules*/
 solve(%p($x)):-equivalent($x,$y),solve(%p($y)).
```

Suppose the initial goal is p(b). p(b) is failed at base level and is derived by reference as a metalevel goal solve(<p>("b")), which is then solved with meta-evaluation rules at metalevel. When the goal is satisfied by the meta-evaluation rule, a substitution {<p>/%p, "b"/$x} is applied and two sub-goals, equivalent("b",$y) and

solve(<p>($y)) are generated. Here the lexical form takes into effect as: %p is predicate-metavar, $v general-metavar, "c", "?v" quoted-namecons, and <pred> bracketed-namecons. See appendix to refer to all substitution rules about these entities.

The first sub-goal is satisfied by the metalevel assertion with a substitution {"a"/$y}, and the second is then evaluated, causing a de-reference to a base level goal p(a), which then satisfied at base level obviously. Consequently, the initial goal will succeed.

4.1 A Kind of Grinding

The power of GOP is that the program can be understood at any granulation levels, without concerning details. As listed in Appendix, two separate programs are grounded into code ingredients: classes, methods, and generic-functions. They are compounded into a class hierarchy (called kernel class hierarchy, KCH) and a set of generic-functions using CLOS (Common Lisp Object System) inheritance mechanism and method combination mechanism[16].

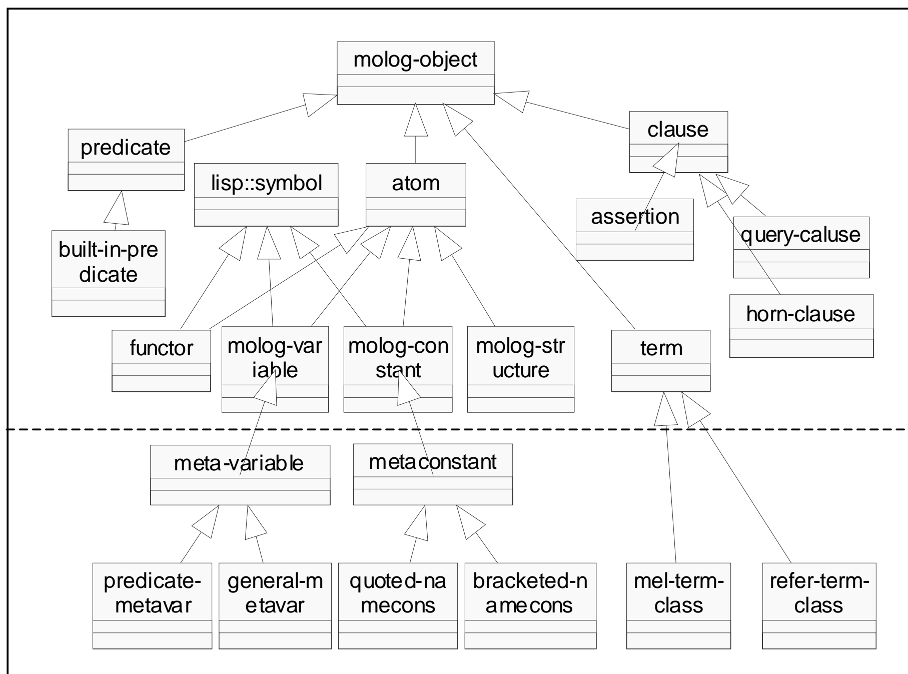

Figure 3. Kernel class hierarchy of PPS0 & PPS1

In figure 3, classes above the dashed line form the KCH of PPS0, and classes under the line are added classes only for PPS1. In fact, we can view the PPS0 as a basic PPS in the domain. Then PPS1 can be simply viewed as the added part as listed in the appendix.

Code ingredients of Lisp program can be defclass form, defmethod form, defgeneric form, define-method-combination form, or even function definition. Here what we are interested in are decided by the goal of grinding. The most code ingredients for this example, we are interested in, have been listed in Appendix, and numbered in order to refer.

CLOS standard method combination mechanism can be viewed as an alternative by which one or more methods are compounded as a generic-function. In this example, we just use *primary* method and *around* method. For instance, code ingredient 40 in appendix is a primary method, 28 is an around method, both to parameter specializer list (term t t). The sequence of execution of both kinds of methods is generally as follows: (*u1, u2, ... , uu*), where *ui*, $1 \leqslant i \leqslant u$, denotes around method in method specific order. If ui calls call-next-method, then it calls the next one in the sequence, or if no one left then calls the follows: *(p1, ..., pp)*, where *pk,* $1 \leqslant k \leqslant p$*,* denote primary methods, in a method specific order. For the above methods, the call-next-method form in code ingredient 28 will call code ingredient 40, implementing the Reflective Prolog's goal derivation as described in the beginning of this section. For details of method combination, please refer to ANSI Common Lisp standard documents[16].

4.2 Granule Compounding

In this example, the compounder is based on CLOS object-oriented facilities such as dynamic sub-classing and runtime method combination. Classes from separate PPS can be compounded using CLOS add class mechanism. Methods from different PPS can be compounded using CLOS add method and method combination mechanism. The coherence between granules is guaranteed manually. The granular output code is gained by merging the initial granular program of PPS1 into that of PPS0.

Some granules should be compounded into a high-level granule in this example. For instance, *object representation*, *variable substitution*, *goal derivation*, etc. These upper-layer granules are corresponding to Prolog mechanisms, and can be interpreted at logic level of Molog system.

5. CODE GRANULATION SPACE

The code granulation space is dedicated to a primary problem solving in a domain, which is an expression of the program in multiple-abstraction framework. The goal of building code granulation space for PPS's is to localize unfitness in a well-formed and multi-layered framework. We believe that for a given domain, the code granulation space of every PPS can be merged into a complete code granulation space of the domain. From this viewpoint, the code granulation space of every PPS is partial space of the complete space the domain has. A programmers' responsibility is to make step forward to the complete space from the current PPS's.

From the first example given in Section 3, we can explain how the program PPS3 is programmed using PPS0 to PPS2 by merging their code granulation spaces together. Granule `<quicksort:>` serves all the three PPS's. We say that its context is more general than others in the Figure 2. Fitness of lower level granules to their context is steadier than that of high-level granules. For example, whenever the context is *context0*, *context1*, or *context2*, granule `<quicksort:>` keeps well fitness to each of the context. In fact, the context the granule `<quicksort:>` can be seen is just the data structure, which is same in every PPS instead of the data structure name. This is a context distribution in the code granulation space. That is, the context of a PPS is distributed as every granule's context in the space. If the program becomes unfitting to its current context, then this will have an impact on the fitness of some granules that we are just want to localize.

The following is some issues about Granule-Oriented Programming.

Similarity detection. The main goal of grinding is to localize unfitness by similarity analysis between primary problem solving cases for a domain. So similar code ingredients is more general than dissimilar code ingredients.

Context distribution. As described above, the goal of context distribution is to determine that each granule in code granulation space has right part of context of the whole PPS. A general idea about context distribution is that each granule is just responsible for fitness to its own context.

Zooming-in/zooming-out. Zooming-in and zooming-out are basic mechanisms to describe how lower level granules are compounded into upper level granules in code granulation space. For granule `<producer:>` in Figure 2 as an example, it is compounded from granule `<read_port:>` and `<mutual_write:>`. The zooming-in/out between lower level {`<read_port:>`, `<mutual_write:>`} and upper level {`<producer:>`} is defined as following code finally:

```
{ int tmp;
  while(!(<read_port:data,port>)){
   <mutual_write:lock,data,buffer>;
  notify();
  }
}
```

In the second example, logic entities such as clause, term, etc can be viewed as granules, which are upper level granules to the objects that represent these logic entities. For zooming-in/out between logic entities and objects, the code 41-45 of PPS0 is derived. And similar code ingredients 25-27 from PPS1 are added in order to new logic entities defined in Reflective Prolog. Zooming-in/out is the basic mechanism to construct code granulation space.

6. RELATED WORK

Lots of existing work appears to be based on intuitions similar to those underlying Granule-Oriented Programming.

Aspect-oriented programming. AOP makes it possible to define additional implementation to run at certain well-defined points in the execution of the program, namely dynamic crosscutting mechanism, which is based on a small but powerful set of constructs[8]. AOP stresses separation of concerns in programming, by means of *advice* and *pointcuts*, a method-like construct, for example, CLOS method combination framework. The goal of AOP is to make it possible to deal with crosscutting aspects of a system's behavior as separately as possible. AOP provides a means of code grinding by which code ingredients can be classified by whether the aspect they are belonging to or not. As the result, these code ingredients can be compounded and form a granulation space where aspects will be granules at some level. Aspectual code granulation is useful to capture the system's behavior from separate concerns in the domain.

AOP supports a significant design philosophy, that is, separation of concerns. However, it pays no attention to the organization of the woven code. You cannot locate which piece of code is corresponding to which aspect that you used in programming stage. GOP supports code granules construct and organize during the whole life cycle of programs. That means the code granulation space is as important as, or even more important than the program itself.

Join points in AOP are well-defined for single-abstraction systems[9], while granules in GOP is a more general code weaving mechanism because granule is a building block instead of a label. Well-defined code granulation space is a multi-abstraction of domain.

Object-oriented programming. OOP provides powerful language constructs for organizing a program as a group of communicating objects. For example, classes, methods, inheritance hierarchy is helpful to describe the system's building blocks and relationship between them; message-passing mechanism is useful to realize behavioral relationship between objects. Classes, methods can be special cases of code ingredients and granules in GOP. And subclassing and message passing can be special cases of granulation.

Reflection. Reflection is a powerful mechanism in programming language to support a program deals with its own facilities in the course of solving domain problems. Reflective programming languages provide language constructs or facilities for programmers to deal with the program's context in more explicitly than non-reflective languages. There are great efforts on reflective language design such as 3-Lisp[15], CLOS metaobject protocol[7], and some work on prolog, Java, Smalltalk, and C++ reflective mechanisms extensions[2][11][14], etc. The program's context is paid more attention to in reflective languages than non-reflective languages. For example, sub-classing mechanisms, method combination frameworks, etc., which are provided by a reflective object-oriented language, can be processed by a program written using exactly the language. So unfitting phenomenon has been explored in some extent with reflective facilities in these languages.

Layered view on a reflective system is helpful to explain how it works. The base-level of the system is what is solving domain problems; while the meta-level of the system is what is solving problems about how the base level works. That means the meta-level is responsible for keeping the system fitting to its context as enough as possible.

The homogeneity of metalevel architecture makes programming easily but what can be programmed on the context is highly restricted. So a variety of unfittings still remain unsolved in the programming due to the fact that you cannot describe, in a language, what is beyond the language's expressivity in the real systems.

Some other languages. Component-based design[17] is a methodology that tries to find the system's common behavior and then generalizes them into the reusable components. Reusable components provide similar functionality as granules to localize special behavior and separate them with the other part of the system. There are some difficulties with reusable components. For example, the methodology faces how to generalize a single component from a special case. In GOP granules are organized into a granulation space, which is more helpful to granule's evolution from one PPS to another because constraints from other granules specialize the granule compared to reusable components.

Generative Programming[4] provides a means for developing programs that synthesize other programs. The goal of generative programming is to replace manual search, adaptation, and assembly of components with the automatic generation and configuration of components on demand. This idea is similar to GOP that a program can be partially derived from the existing programs. Generative programming is based on assembly of components. In GOP we pay attention to the similarities of primary problem solving cases and interpret them in every granulation levels.

7. CONCLUSIONS

Invariable agreement between program and its context will cause either a premature end of the program's life cycle, or relatively decreasing the program's effectiveness. The case that a program becomes unfitting to its context exists in

many complex software systems. When programmers force a system to solve new problems in the domain, unfitting phenomenon may occur. When the system is deployed on a different runtime environment, this phenomenon may occur.

Granule-oriented programming is an evolvement metaphor in which programs are "ground" into code ingredients in order to locate unfitting part of programs as explicitly as possible, and then "compound" into granular output code in which the code granulation space is formed. Code granulation space is an expression of a program in multiple-abstraction framework. The goal of building code granulation space for a program is to localize unfitness in a well-formed and multi-layered framework. Zooming-in and zooming-out between multiple layers in the granulation space are helpful to explore the unfitting phenomenon.

Code granules evolve from one primary problem solving case to another in a domain. We can conclude that evolved code granules are more generic, that is, they have been reused by more primary problem solving cases in the domain. Case studies on granule-oriented programming show that the fitness of a program to its context can be expressed with the fitness of the program granules to their contexts, respectively.

Some of future directions of GOP are: application of GOP technology, conceptual foundations for understanding unfitness and compounding, foundations for code granule space and context distribution, development of GOP toolkit technology, etc.

8. ACKNOWLEDGMENTS

This report is written during my visit at Department of Computer Science, University of Regina (April 6th, 2003 to April 5th, 2004). I wish to thank Department of Computer Science, Faculty of Science, and University of Regina for providing the opportunity and partial support. I wish to thank Xi'an Jiaotong University for providing the opportunity and partial support, too. I thank Dr. Y.Y. Yao and Dr. J.T. Yao for helpful discussion on the concept of granular computing, and thank to Baohong Li, Yongzhong Zhang, and Bo Liu for working on some GOP examples.

9. REFERENCES

[1] Chaudron, M. R. V. and Laar, F.V. An upgrade mechanism based on publish/subscribe interaction. In Workshop on Dependable On-line Upgrading of Dist. Systems in conjunction with COMPSAC 2002, (Oxford, England), Aug. 2002.

[2] Costantini, S. Meta-reasoning: a survey. In: Robert, A. et.al. (eds.) Computational Logic: Logic Programming and Beyond, Lecture Notes in artificial Intelligence 2407-2408, Springer-Verlag, 2002. 63pages

[3] Costantini,S. and Lanzarone,G.A., Metalevel Representation of Analogical Inference, LNAI549, 1991, 460-464

[4] Czarnecki, K. and Eisenecker, U.W. Generative programming – methods, tools, and applications, Addision-Wesley, 2000

[5] Foster, I. Designing and building parallel programs. Addison-Wesley, 1995

[6] Jackson, P., Reichgelt, H. and Harmelen, F.V. (eds.) Logic-based knowledge representation, MIT Press, 1989.

[7] Kiczales, G. Rivieres, J. and Bobrow, D.G. The Art of the Metaobject Protocol. MIT Press, Cambridge, 1991

[8] Kiczales, G. Hilsdale, E., Hugunin, J., Kerstn, M., Palm, J. and Griswold, W. An Overview of AspectJ. In proc. European Conference on Object-Oriented Programming, 2001, Springer-Verlag LNCS 2072.

[9] Kiczales, G. Lamping, J., Mendhekar, M., Maeda, C., Lopes, C., Loingtier, J-M and Irwin, J. Aspect-Oriented Programming. In proc. European Conference on Object-Oriented Programming (ECOOP'97), Springer-Verlag LNCS 1241, 1997, 220-242

[10] Lopes, C.V., Dourish, P., Lorenz, D.H. and Lieberherr, K. Beyond AOP: Toward naturalistic programming, in ACM OOPSLA'03 proceedings (October, 2003, California), 198-207

[11] Maes, P. Concepts and Experiments in Computational Reflection. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Orlando, Florida. (1987), 147-155

[12] Nolte, J. Language level support for remote object invocation. Technical report, GMD FIRST, Berlin, 1991

[13] Rich, E. and Knight, K. Artificial intelligence, McGraw-Hill, 1991.

[14] Richmond, M. and Noble, J. Reflections on remote reflection. Proceedings of the 24th Australasian conference on Computer science, Volume 23 Issue 1. (2001), 163-170

[15] Smith, B.C. Reflection and Semantics in LISP. In Proceedings of the Symposium on Principles of Programming Languages (POPL). ACM. (1984) 23-35

[16] Steele, G.L. Common Lisp the Language, 2nd edition, Digital Press, 1990

[17] Szyperski, C. Component Software – Beyond Object-Oriented Programming, 2nd Ed., Addison-Wesley, ACM Press, 2002

[18] Zhao, Y.L. Zheng, S.Q. A logical meta-level architecture based on meta-objects. in Technical Digest of International Symposium on Information Science and Technology, International Academic Publishers (Beijing, 1996), 253-256.

APPENDIX: INTERESTING CODE INGREDIENTS OF MOLOG EXAMPLE

```
;;;PPS1 code ingredients
1 (defclass metavariable (molog-variable)  ()
  (:metaclass molog-atom-class))
2 (defclass predicate-metavar (metavariable)
  ((value :accessor predicate-metavar-name))
  (:metaclass molog-atom-class))
3 (defclass general-metavar (metavariable)
  ((value :accessor general-metavar-name))
  (:metaclass molog-atom-class))
4 (defclass metaconstant (molog-constant)  ()
  (:metaclass molog-atom-class))
5 (defclass quoted-namecons (metaconstant)
  ((value :accessor quoted-namecons-value))
  (:metaclass molog-atom-class))
6 (defclass bracketed-namecons (metaconstant)
  ((value :accessor bracketed-namecons-value))
  (:metaclass molog-atom-class))
7 (defmethod var-replace
       ((x predicate-metavar) (y bracketed-namecons) env)
  (acons x y env))
8 (defmethod var-replace
       ((y bracketed-namecons) (x predicate-metavar) env)
  (acons x y env))
9 (defmethod var-replace
       ((x predicate-metavar) (y metavariable) env)
  (acons x y env))
10(defmethod var-replace
       ((y metavariable) (x predicate-metavar) env)
  (acons x y env))
11(defmethod var-replace ((x predicate-metavar) y env)
  (declare (ignore y env))   nil)
12(defmethod var-replace (y (x predicate-metavar) env)
  (declare (ignore y env))   nil)
13(defmethod var-replace ((x general-metavar) (y metaconstant) env)
  (acons x y env))
14(defmethod var-replace ((y metaconstant) (x general-metavar) env)
  (acons x y env))
15(defmethod var-replace ((x general-metavar) (y metavariable) env)
  (acons x y env))
16(defmethod var-replace ((y metavariable) (x general-metavar) env)
  (acons x y env))
17(defmethod var-replace ((x general-metavar) y env)
  (declare (ignore y env))   nil)
18(defmethod var-replace (y (x general-metavar) env)
  (declare (ignore y env))   nil)
19(defmethod var-replace
       ((x quoted-namecons) (y quoted-namecons) env)
  (var-replace (quoted-namecons-value x)
             (quoted-namecons-value y)   env))
```

```
20(defmethod var-replace
       ((x bracketed-namecons) (y bracketed-namecons) env)
  (var-replace (bracketed-namecons-value x)
             (bracketed-namecons-value y)  env))
21(defmethod var-replace ((x metaconstant) y env)
   (declare (ignore y env))    nil)
22(defmethod var-replace (y (x metaconstant) env)
   (declare (ignore y env))    nil)
23(defclass mel-term-class (term) ())
24(defclass refer-term-class (term) ())
25(defmethod make-molog-atom :around ((token string))
  (let* ((tag (char= (char token 0) #\?))
         (val (read-from-string token)))
     (cond (tag (make-instance 'quoted-namecons
                 :value (make-instance 'molog-variable :value val)))
      ((symbolp val)
        (make-instance 'quoted-namecons :value val))
      (t val))))
26(defmethod make-molog-atom :around ((token symbol))
   (let ((object (call-next-method)))
     (if (eq (class-of object) (find-class 'symbol))
     (let* ((sym object)
         (leading-char (char (symbol-name sym) 0)))
       (cond ((char= leading-char #\$)
                (make-instance 'general-metavar :name sym))
               ((char= leading-char #\%)
                (make-instance 'predicate-metavar :name sym))
               ((char= leading-char #\<)
                (make-instance 'bracketed-namecons
                    :value (intern (string-left-trim '(#\<)
                           (string-right-trim '(#\>) sym))
                                *package*)))
               (t object)))
        object)))
27(defmethod make-term :around (name arg-list)
   (if (eq name 'solve)
       (change-class (call-next-method) 'mel-term-class)
       (call-next-method)))
28(defmethod satisfy-goal :around ((goal term) env cont)
  (let ((class-name (class-name (class-of goal))))
    (ecase class-name
      (term (call-next-method)
       (satisfy-goal (reference goal) env cont))
      (mel-term-class
        (multiple-value-bind (new-goal new-env)
      (dereference goal env)
      (satisfy-goal new-goal new-env cont)))
      (refer-term-class
       (call-next-method)))))
29(defun reference (goal)
 (when (y-or-n-p) (throw 'end-query nil))
  (let ((comps (mapcar #'(lambda(arg)
```

```
                    (make-instance 'quoted-namecons
                              :value arg))
                (term-arg-list goal)))
    (name (make-instance 'bracketed-namecons
            :value (functor-name (term-functor goal)))))
   (make-instance 'refer-term-class
      :arg-list (list (make-instance 'molog-structure :name name
                       :components comps))
      :functor (make-instance 'functor :name 'solve :arity 1))))
30(defun dereference (goal env)
  (flet ((get-val (x)
      (let ((val (value-of x env)))
        (when (and (typep x 'general-metavar)
                (typep val 'quoted-namecons)
                (typep (quoted-namecons-value val)
                      'molog-variable))
           (setq env (acons x (make-instance 'quoted-namecons
              :value (value-of (quoted-namecons-value val) env))
                     env)))
           val)))
  (let* ((arg (car (term-arg-list goal)))
          (functor-name (value-of (molog-structure-name arg) env))
          (arglist (mapcar #'get-val
                 (molog-structure-components arg)))
          (new-arglist nil))
     (setq functor-name (bracketed-namecons-value functor-name))
     (dolist (arg arglist)
       (multiple-value-bind (var new-env) (dereference-atom arg env)
          (setq new-arglist (cons var new-arglist))
          (setq env new-env)))
      (setq arglist (reverse new-arglist))
      (values
       (make-instance 'refer-term   :arg-list arglist
       :functor (make-instance 'functor :name functor-name
                 :arity (length arglist)))
   env))))
31(defmethod dereference-atom(x env)
   (values (molog-atom-value x) env))
32(defmethod dereference-atom((x metavariable) env)
   (let ((new-var (make-instance 'molog-variable
           :name (molog-variable-name x))))
    (values new-var
          (acons x (make-instance 'quoted-namecons :value new-var)
              env))))
33(defmethod print-object ((v quoted-namecons) strm)
  (format strm "~s" (format nil "~S" (quoted-namecons-value v))))
34(defmethod print-object ((v bracketed-namecons) strm)
  (format strm "~a"
        (format nil "<~S>" (bracketed-namecons-value v))))
35(defmethod print-object ((v predicate-metavar) strm)
  (format strm "~s" (predicate-metavar-name v)))
36(defmethod print-object ((v general-metavar) strm)
```

```
  (format strm "~s" (general-metavar-name v)))

;;; Part of PPS0 code ingredients
37(defmethod prove ((clause query-clause))
  (catch 'end-query
    (let* ((var-bindings (make-new-names clause))
          (goals (clause-body (instantiate clause var-bindings))))
      (satisfy-goal-list goals '(nil) #'print-solution)
      t)))
38(defmethod satisfy-goal-list ((goals null) environment continuation)
    (funcall continuation environment))
39 (defmethod satisfy-goal-list ((goals list) env cont)
  (satisfy-goal (car goals) env
          #'(lambda (env1) (satisfy-goal-list (cdr goals) env1 cont))))
40(defmethod satisfy-goal ((goal term) env cont)
  (try-predicate (term-predicate goal) goal env cont))
41(defmethod make-term ((functor-name symbol) arg-list)
  (if (eq functor-name '!)
      (make-instance 'term
       :functor (make-instance 'functor :name functor-name
                    :arity (length arg-list))
       :arg-list (list *cut-host*))
      (make-instance 'term
       :functor (make-instance 'functor :name functor-name
                    :arity (length arg-list))
        :arg-list arg-list)))
42(defmethod make-molog-atom ((token number))  token)
43(defmethod make-molog-atom ((token string))  token)
44(defmethod make-molog-atom ((token symbol))
  (if (eq token '?) *anonymous-var-obj*
      (let ((leading-char (char (symbol-name token) 0)))
      (if (char= leading-char #\?)
          (make-instance 'molog-variable :value token)
          token))))
45(defmethod make-molog-atom (object)
  (molog-error "Unrecognized data object ~a" object))
46(defmethod var-replace ((x molog-variable) y env)  (acons x y env))
47(defmethod var-replace (x (y molog-variable) env)  (acons y x env))
48(defmethod var-replace (x y env)  (if (equal x y) env nil))
49(defmethod var-replace ((x molog-constant)(y molog-constant) env)
  (if (equal (molog-constant-value x) (molog-constant-value y))
      env nil))
50(defmethod var-replace ((x molog-structure) (y molog-structure) env)
  (unify-aux
     (cons (molog-structure-name x) (molog-structure-components x))
     (cons (molog-structure-name y) (molog-structure-components y))
     env))
51(defmethod var-replace ((x anonymous-var) (y molog-atom) env)  env)
52(defmethod var-replace ((x molog-atom) (y anonymous-var) env) env)
```