

MODELING ANALOGICAL REASONING WITH DYNAMIC INHERITANCE SYSTEM IN C++

Oleg Ostrozhansky
Department of Computer Science
Illinois Institute of Technology
e-mail: thssoxo@iitmax.acc.iit.edu
December 1994

A **rose** is a garden flower which has a lot of petals and a pleasant smell.

(English Language Dictionary)

Abstract

Dynamic Inheritance System (DIS) allows creations of objects and IS-A links among them at run time. This enables artificial intelligence programs modeling human reasoning process to develop and use a hierarchical view of the world, separating concepts from details of specific situations.

Introduction

The importance of inheritance is nothing new. Object oriented approach to programming has proven to be superior to other traditional approaches in many areas. However, the concept of one entity being a specialized instance of, or inheriting from another entity is much more than just another approach to writing computer programs. It is one of the key concepts in our understanding of the world. Two examples illustrate this point. The first one is an English language dictionary where most definitions are in a form "X is Y where [where clause]", meaning that X inherits all properties of Y except for, or in addition to the properties mentioned in the where clause. The second example is a child who sees an unfamiliar object and asks a parent: "Mom, what is it?" implying that the object must be placed somewhere in the inheritance structure in his mind with an IS-A relationship.

Dynamic Inheritance System for C++ Programs

What is DIS

The Dynamic Inheritance System (DIS) allows a class hierarchy to be built at run time. This means that the class hierarchy does not need to be created and finalized before a program is compiled, but it can be modified, or even created from scratch, during

execution of a program. This allows dynamic systems like a neural network or a genetic algorithm to modify the inheritance used in a program, opening doors for such algorithms to model human understanding of the world and process of human learning and decision making.

How to use DIS

DIS provides a set of functions that can be called from a C++ program to create, modify, and search a class hierarchy. The functions are (see DIS.H in Appendix A for complete function declarations):

```
Object *new_object(char *object_name)
Object::set_superclass(Object *super)
Object::rm_superclass(Object *super)
Object::set_attrib(char *attrib_name, void *attrib_value)
Object::rm_attrib(char *attrib_name)
Object::get_attrib(char *attrib_name, Object **where)
```

The basic entity in DIS is an **Object**. Objects have **Attributes**. Each Attribute has a name and a value. An attribute's name is a character string, and its value is a generic pointer. Attributes are accessed using an Object and Attribute's name. When retrieving the value of an Attribute, if the Attribute is not found in the Object, then Objects that the current Object inherits from (ancestors) are searched, and the first match of the Attribute is returned. First the direct ancestors are searched (the ones the current Object inherits from directly), then their direct ancestors, and so on. When searching direct ancestors of an Object, they are searched in the reversed order in which they were added.

The last argument of `get_attrib` returns a pointer to the Object where the attribute returned is defined. If this information is not needed, a NULL can be passed as the last argument. Using the keyword **PRIVATE** in place of the last argument limits the search for an attribute to searching the current object only.

For any two Objects one can always be made a direct ancestor of the other, even if they are already related. Thus DIS allows multiple inheritance and even cycles. Any child-ancestor link can be removed at any time.

Although an Attribute's value is a generic pointer, it can be used to store anything, including a pointer to a function, allowing Objects to have *methods*, or defined operations, without DIS having to do anything special for it.

Users use class Object to create a C++ object for each DIS Object, and then use member functions of Object to create and remove inheritance links between Objects, set and remove Attributes, and traverse the inheritance structure.

For example, the following code creates an object and adds attributes to it:

```
dog = new_object("dog");
dog.set_attrib("has_tail", "yes");
dog.set_attrib("color", "grey");
```

The following creates another object and makes it a subclass of the previously created object `dog`:

```
poodle = new_object("poodle");
poodle.set_attrib("color", "black");
poodle.set_superclass(dog);
```

Getting `poodle`'s attributes `"has_tail"` and `"color"` with `get_attrib()` works as expected, returning `"yes"` and `"black"` respectively.

DIS Implementation

DIS is implemented using C++ and the object oriented paradigm. Note, however, that inheritance built in DIS has nothing to do with inheritance in C++.

Each Object has, among other things, a set of Attributes, a set of pointers to Objects it inherits from (direct ancestors, or direct superclasses), and a set of pointers to Objects that inherit from it (direct children, or direct subclasses). All sets are managed with a Queue class. When an Attribute is assigned a value, it is added to the set of Attributes of the Object if there is no attribute with the same name in this Object, or the value of an existing Attribute is changed if there is. Each Attribute remains in the Object until it is explicitly removed with `rm_attrib()`.

Retrieving a value of an attribute requires traversing the inheritance structure. A temporary Queue is used to keep track of Objects that will need to be searched. To avoid searching the same Object multiple times, a field *visited* (of *long* data type) is used. Each time the inheritance structure is traversed, a static counter *visited_cnt* is incremented, and *visited* of all Objects being searched is assigned *visited_cnt*. Objects with *visited* matching *visited_cnt* can then be ignored because they have already been searched during this inheritance traversal. This setup avoids going through *all* Objects at any time and is guaranteed to work at least 4,294,967,295 times. And if this becomes a limitation, all that is needed is to go through all objects once in that number of inheritance searches and reset *visited* of all Objects to 0. (After all, if a computer is used to simulate a human activity as DIS is intended to, there is nothing wrong with it requiring to sleep sometimes.)

Sets management: supporting class QUEUE

The class Queue implements a doubly linked queue of anything, and is used to manage all sets, or collections of any objects, in DIS. The following functions are provided by QUEUE:

```
QUEUE::insert_first(void *data)
QUEUE::insert_last(void *data)
QUEUE::remove(void *data)
```

A structure `Q_LINK` is used to store pointers to previous and next elements and the pointer to data. The user can (and must) search the QUEUE elements directly, but must use QUEUE member functions to change anything in the QUEUE.

Memory management: supporting class FREE_LIST

A class FREE_LIST is used for all memory management needs. The idea is that when frequent allocations and deallocations of elements of the same type are required, it is best to allocate them in chunks when needed, and instead of returning them back to the Operating System when no longer needed, keep them for the next allocation requests. This is exactly what FREE_LIST does. FREE_LIST has a pointer to a chain of free structures, and allocation requests are satisfied from that chain. Structures no longer needed are returned back to the chain. Pointers that link the chain are overlaid with the structures themselves, as their contents are not needed when the structures are under FREE_LIST control. To satisfy an allocation request when no more structures are left in the chain, a new chunk (of user defined number) of structures is allocated from the Operating System, linked into a chain, and one of the newly allocated structures is returned. For example, Q_LINK structures are used extensively by the QUEUE class. Thus, it contains a declaration:

```
FREE_LIST free_q_links(sizeof(Q_LINK), 200)
```

creating FREE_LIST object for managing Q_LINK structures which allocates new structures in chunks of 200. When a Q_LINK structure is needed, it is gotten by

```
free_q_links.allocate()
```

When no longer needed, it is deallocated by

```
free_q_links.free()
```

This way of memory management has proven to be very efficient in DIS and many other programs with similar memory management needs. It effectively makes resources required to allocate and deallocate structures negligible as most requests are satisfied with only a few simple pointer manipulations, does not need a garbage collector, and does not experience any memory leaks.

Examples of DIS usage

A simple example (EXAMPLE.CPP)

This example illustrates how the functions provided by DIS are to be used. Appendix E provides a complete listing and output, which are self-explanatory.

Prototype layer (PROTOT.CPP)

The file PROTOT.CPP implements another layer between DIS and a user program. It provides a quick way to create methods and test their activations, and is intended for quick prototyping of programs using DIS. The target is to eliminate the necessity of creating a separate function in the user program for each method in the inheritance hierarchy. Examples of methods creations of which are very much simplified by the prototype layer are methods which only call other methods, and methods which only print a message.

It also illustrates how DIS can be extended to handle any type of data. The program in `OBSTACLE.CPP` described later uses this layer.

The prototype layer includes the following functions:

```
func(fn, arg, flags)
```

which creates a structure to be used by `invoke` to execute function `fn` as a method of an Object. When used as

```
obj.set_attrib(meth_name, func(fn, arg, flags))
```

it creates a method `meth_name` in object `obj` which is implemented by function `fn`. The function `fn` will be called by `invoke` with argument `arg` if method `meth_name` is invoked (either on the object `obj` itself, or on any of its children who do not have the same method redefined). If `flags` include `F_OBJ_PTR`, then `invoke` will call function `fn` with a pointer to the current object as the first parameter. If `fn` is `NULL`, `arg` will be taken as the result of the method. If `flags` include `F_GO_ON`, the result of the method will be treated as a name of a function to be executed (or a sequence of functions to be executed with their names separated by spaces, i. e. a *script*) on the current object. The flags can be OR-ed together.

```
invoke(obj, meth_name)
```

executes a method named by `meth_name` in the Object `obj`.

```
rm_func(f_struct)
```

removes the structure created by `func()`.

Examples of the prototype layer usage:

```
obj.set_attrib("print_hello", func(sprintf, "hello", 0));
invoke(obj, "print_hello");
```

When `print_hello` method is invoked, `printf` is called with "hello" as the only argument, and "hello" will be printed. The alternative which does not use the prototype layer is:

```
// function print_hello prototype
void print_hello(void);
:
// add attribute pointing to the print_hello function
obj.set_attrib("print_hello", (void *)print_hello);
// call the function pointed to by "print_hello" attribute
((void (*)(void))(obj.get_attrib("print_hello", NULL))());
:
// function print_hello implementation
void print_hello()
{
    printf("hello");
}
```

Another example:

```
obj.set_attrib("say_hello", func(NULL, "print_hello",
    F_GO_ON));
invoke(obj, "say_hello");
```

The `invoke()` call here assumes that there is a method `print_hello` added using the prototype layer functions, and this `print_hello` method is called.

And to remove a method created using the prototype layer:

```
if ((f = obj.get_attrib("say_hello", PRIVATE)) != NULL) {
    rm_func(f);
    obj.rm_attrib("say_hello");
}
```

This removes the method `say_hello` from Object `obj`.

See `OBSTACLE.CPP` for more examples of the prototype layer usage.

Models of Analogical Reasoning Using DIS

DIS is intended for use with systems that require modifications to the class hierarchy "on the fly". Examples of such systems are programs trying to simulate human reasoning and decision making process by utilizing the inheritance structure that I think is created in the human mind (as it has been described in the **Introduction**). Here is how DIS could be used by these programs (in fact, this is what DIS was originally intended for).

Running Into an Obstacle (OBSTACLE.CPP)

The situation here is a computer program simulating a person solving a mathematical problem. A person tries to apply simple methods first, does it for some time, and then realizes that the problem is more complicated than he expected. At this point the problem in his mind becomes associated with an *obstacle* and he knows that to deal with an obstacle he needs either to work harder or invent something different (metaphorically *go around*). With DIS it means that there is an Object *Obstacle* somewhere, which becomes a superclass of the mathematical problem at hand (the problem becomes an obstacle). This *Obstacle* object contains the knowledge related to all obstacles, that is the existence of two alternatives for overcoming an obstacle: *try harder* and *go around*. However, both *try harder* and *go around* then must be instantiated with respect to the problem at hand, suggesting that the current object must have specialized versions of *try harder* and *go around*. Thus, when a mathematical problem becomes an obstacle, *try harder* can get translated into integrating, and *go around* may be reading a textbook. The program `OBSTACLE.CPP` does exactly this using DIS. A method `do` of `my_diff_problem` is repeatedly activated, but what it translates into depends on the inheritance structure built in DIS. As inheritance structure is modified when `my_diff_problem` becomes a subclass of *Obstacle*, `do` method of `my_diff_problem` activates the `do` method in the *Obstacle* which decides what to do with the problem that has become an obstacle. The inheritance structure for this example is shown in the Fig. 1. Note that exactly the same mechanism,

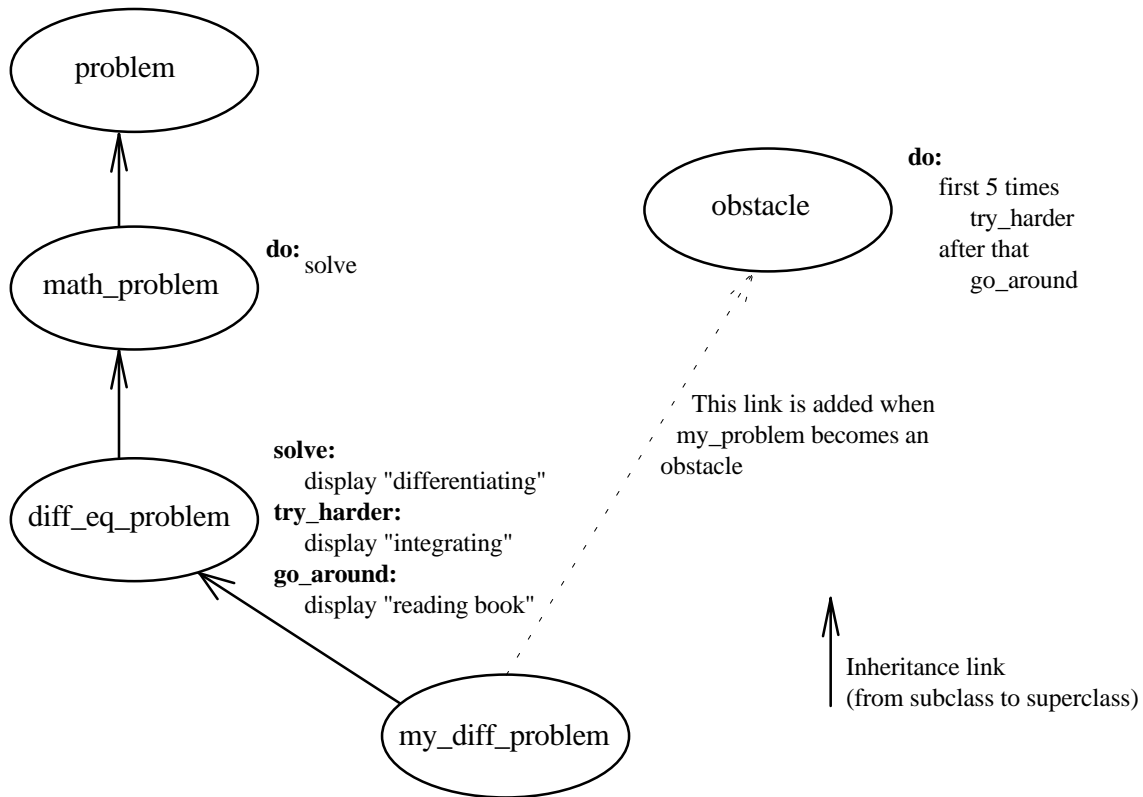


Fig. 1 Inheritance structure in OBSTACLE.CPP

with the same *Obstacle* object, will work in many other situations which are all classified in a metaphorical sense as running into an obstacle.

The key line in this example is

```
my_diff_problem->set_superclass(obstacle);
```

Since the *obstacle* object has a definition for the method *do*, there is no specific *do* in *my_diff_problem*, and the inheritance link between *my_diff_problem* and *obstacle* is the most recent one added, the *do* method of *obstacle* object is the one activated by *engine* as the *do* method of *my_diff_problem*. And the *do* method in the *obstacle* object has all the high level knowledge related to the *obstacle* concept.

Although in *OBSTACLE.CPP* it is programmed that the first 5 attempts to solve the problem fail before the problem becomes an *Obstacle*, in a "real simulation" this will be the job of a neural network or another Artificial Intelligence algorithm. The regular *do* (the one activated when the current problem is not an *Obstacle*) will result in some progress that will be monitored by a neural network. When this progress stops in spite of the continuing efforts, the neural network will associate the *Obstacle* Object with the current problem by adding an IS-A relationship between them using DIS.

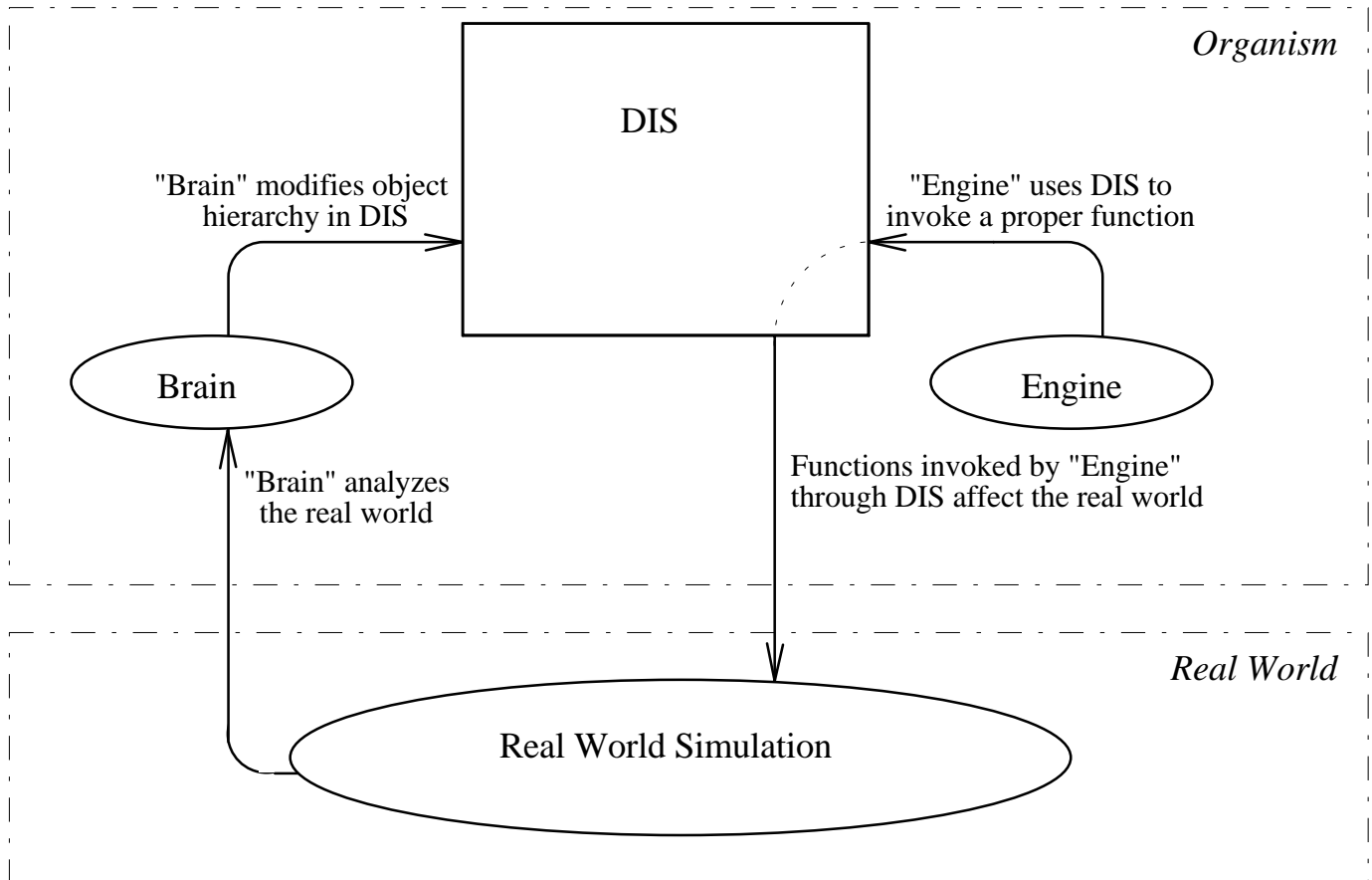


Fig. 2 Proposed organization of an Artificial Intelligence program using DIS

The above example suggests the organization shown in Fig. 2. A *real world simulation* simulates some part of a real world that the simulated organism lives in. The organism can examine the state of the real world, and its actions affect the simulation. The *brain* part of the organism modifies the inheritance structure in DIS. The *engine* part invokes a `do` method using DIS which results in invocation of a proper function, which will modify the *real world simulation*.

Adjusting a Quantitative Value (ADJUST.CPP)

This example has to do with such concepts as *too much*, *not enough*, *doing more*, and *doing less*, which apply to any task involving a single quantitative value affecting the result. The goal is to throw a ball and make it travel exactly the specified distance. The quantitative value that needs to be adjusted to achieve the goal is the effort applied during throwing.

This simulation follows the structure shown in the Fig. 2. When the first attempt does not succeed, *brain* decides that the task involves adjusting a quantitative value, and adds an inheritance link from the current task (`throw_ball_now` object) to the

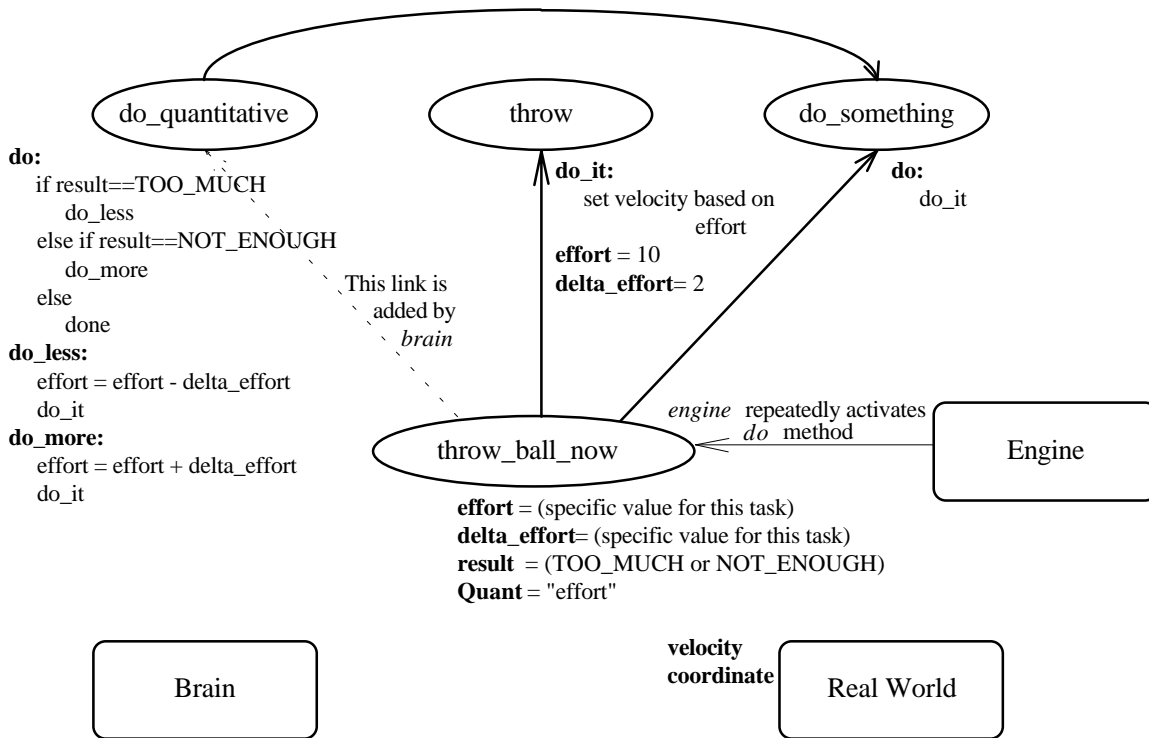


Fig. 3 Inheritance structure and other components in ADJUST.CPP

`do_quantitative` object, making `do_more` and `do_less` methods available. These methods are activated based on the result of the last attempt which is expressed in terms of `TOO_MUCH` and `NOT_ENOUGH`. The process of throwing, implemented in the `do_it` method of the `throw` object, sets *velocity* in the *real world simulation*. *real world simulation* outputs the *coordinate* where the ball landed, based on the *velocity*. *brain* monitoring the real world activity estimates the resulting *coordinate*, decides whether it is `TOO_MUCH` or `NOT_ENOUGH`, and sets the *result* attribute to indicate its decision. The attribute `Quant` with value "effort" points the `do_quantitative` object to `effort` attribute specifying that `effort` is the value to be adjusted. Fig. 3 illustrates the arrangement described above.

Since the estimation performed by *brain* seems a general high level concept that can be reused in many other situations, it is encapsulated in the *estimation* object which is made a superclass of the current task when estimation is needed. It then reacts to *brain's* activation of method `think`.

Conclusion

The work presented here illustrates how inheritance can be used in modeling human understanding of the world, reasoning and decision making process, making possible separation of high level concepts from details of specific situations. The models described provide set-ups for programs modeling analogical reasoning and metaphorical thinking, and facilitate development of such programs. One of them, a feeling-based system, is being developed right now by Greg Chien.

The fact that DIS is implemented in C++ and can be used from C/C++ programs makes it efficient and easy to use in combination with many existing artificial intelligence programs.

References

1. Greene, P. H. *Cognitive structures in movement*. Cahiers de psychologie cognitive, 1987, 7:163-166.
2. Greene, P. H. *The organization of natural movement*. Journal of Motor Behavior, 1988, 20:180-185.
3. Greene, P. H. and Solomon, D. *A computer system for movement schemas*. In Making it Move: Mechanics and Animation of Articulated Figures, ed by Badler, N. I., Barsky, B. A., and Zeltzer, D., pp. 193-205. Palo Alto: Morgan Kaufmann, 1991.
4. Greene, P. H. and Chien, G. T. H. *Feeling Based Schemas*. In Neural Architectures and Distributed AI: From Schema Assemblages to Neural Networks, Center for Neural Engineering Tech rep., U of Southern California, 1993.
5. Greene, P. H. and Kozak, M. *A body-based model of the world*. Proc 5th IEEE International Symposium on Intelligent Control, pp. 193-205. 1990.
6. Winston, P. H. *Artificial Intelligence*, 3rd Edition. Addison-Wesley Publishing Co., pp. 179-197. 1992.

Appendix A. DIS interface, DIS.H

```

/*                      DIS.H

    Dynamic Inheritance System

    Written by Oleg Ostrozhansky, August 1994
*/

#ifndef _DIS_DEFINED_
#define _DIS_DEFINED_

#include <stdio.h>
#include "queue.h"

#define NAMESIZE 15

#define DIS_FUNC(x)  ((int (*)(Object *, void *))(x))
#define DIS_FUNC2(x) ((int (*)(Object *, void *, void *))(x))

// forward references
class Object;

// attribute type
typedef enum {A_NONE, A_REGULAR, A_NEW} A_type;

extern Object **PRIVATE;
    /* for use with Object::get_attrib: if PRIVATE is specified,
       only searches attributes set for this particular instance */

class Attribute {
public:
    char name[NAMESIZE];
    void *value;
};

/***** Object *****/

Object *new_object(const char *class_name);

class Object {
friend Object *new_object(const char *class_name);
public:
    // functions
    Object(); // constructor
    Object(const char *name); // constructor
    void set_superclass(Object *);
    void rm_superclass(Object *);
    A_type set_attrib(const char *name, void *value);
    void rm_attrib(const char *attrib_name);
    void * get_attrib(const char *attrib_name, Object **where = NULL);
    /* returns value of an attribute and shows in which object it was found
       if where == PRIVATE searches only attributes set for this Object
    */

    // variables
    char name[NAMESIZE];

```

Modeling Analogical Reasoning with DIS

```
private:
  // functions
  void init();
  Attribute *find_attrib(const char *name, Object **where = NULL);
  // finds attribute and class to which it belongs
  // variables
  QUEUE attrib;
  QUEUE parents;      // superclasses
  QUEUE children;    // subclasses
  long visited;
};

#define new_class new_object
  // for compatibility with earlier versions

#endif
```

Appendix B. DIS implementation, DIS.CPP

```

/*                                DIS.CPP

    Dynamic Inheritance System

    Written by Oleg Ostrozhansky,  August 1994
*/

#include <string.h>
#include <stdlib.h>
#include "freelist.h"
#include "queue.h"
#include "dis.h"

long visited_cnt = 0L;

static Object *dummy;
Object **PRIVATE = &dummy;

FREE_LIST free_classes(sizeof(Object), 20);
FREE_LIST free_attributes(sizeof(Attribute), 100);

/***** Object *****/

Object::Object()
{
    init();
}

Object::Object(const char *obj_name)
{
    init();
    strcpy(name, obj_name);
}

void Object::init()
{
    attrib.init();
    parents.init();
    children.init();
    visited = 0L;
}

void Object::set_superclass(Object *cls)
{
    parents.insert_first(cls);
    cls->children.insert_first(this);
}

void Object::rm_superclass(Object *cls)
{
    parents.remove(cls);
    cls->children.remove(this);
}

```

Modeling Analogical Reasoning with DIS

```
Attribute *Object::find_attrib(const char *name, Object **where)
{
    static QUEUE search_q;
    static int first_call=1;

    // search attributes of this class
    {
        Q_LINK *q=attrib.head;

        while (q!=NULL && strcmp(((Attribute *) (q->data))->name, name))
            q=q->next;

        if (q!=NULL) {
            if (where != NULL && where != PRIVATE) *where = this;
            first_call = 1;
            while (search_q.head != NULL)
                search_q.remove(search_q.head->data);
            return (Attribute *)q->data;
        }

        if (where == PRIVATE) return NULL;

    }

    // search up the hierarchy, breadth first

    if (first_call) {
        if (search_q.head != NULL) {
            puts("Error: search queue not empty");
            exit(1);
        }
        visited_cnt++;
        first_call = 0;
    }

    visited = visited_cnt;

    for (q=parents.head; q!=NULL; q=q->next)
        if (((Object *) (q->data))->visited != visited_cnt)
            search_q.insert_last(q->data);
}

if (search_q.head != NULL)
    return (((Object *) (search_q.remove(search_q.head->data)))->
        find_attrib(name, where));
else {
    first_call = 1;
    if (where != NULL) *where = NULL;
    return NULL;
}
}

void *Object::get_attrib(const char *name, Object **where)
{
    Attribute *a;

    a = find_attrib(name, where);
    return a==NULL? NULL : a->value;
}

A_type Object::set_attrib(const char *name, void *value)
{
    Q_LINK *q=attrib.head;
    Attribute *a;
```

Modeling Analogical Reasoning with DIS

```
while (q!=NULL && strcmp(((Attribute *) (q->data))->name, name))
    q=q->next;

if (q!=NULL) {
    ((Attribute *) (q->data))->value = value;
    return A_REGULAR;
}

a = (Attribute *)free_attributes.allocate();
strcpy(a->name, name);
a->value = value;
attrib.insert_last(a);
return A_NEW;
}

void Object::rm_attrib(const char *name)
{
    Attribute *a;

    if ((a=find_attrib(name, PRIVATE)) != NULL)
        attrib.remove(a);
}

Object *new_object(const char *name)
{
    Object *c = (Object *)free_classes.allocate();
    c->init();
    strcpy(c->name, name);
    return c;
}
```

Appendix C. Queues management, QUEUE.H and QUEUE.CPP

QUEUE.H

```
/*                                QUEUE.H

    QUEUE class

    maintains a doubly-linked list of whatever

    Written by Oleg Ostrozhansky, December 1993
*/

#ifndef _QUEUE_DEFINED_
#define _QUEUE_DEFINED_

class Q_LINK {
public:
    Q_LINK *next, *prev;
    void *data;
};

class QUEUE {
public:

    // functions
    QUEUE();                                // constructor
    void init();

    // all these functions return pointer to the argument
    void *insert_first(void *);            // insert in the beginning
    void *insert_last(void *);            // insert at the end
    void *remove(void *);                  // removes this element

    Q_LINK *head, *tail;
};

#endif
```

QUEUE.CPP

```
/*                                QUEUE.CPP

    Implementation of QUEUE class

    Written by Oleg Ostrozhansky, December 1993.
    August 94, modified to use Q_LINK structures
*/

#include <stdlib.h>
#include "queue.h"
#include "freelist.h"

static FREE_LIST free_q_links(sizeof(Q_LINK), 200);

QUEUE::QUEUE()
{
    init();
}

void QUEUE::init()
{
    head = tail = NULL;
}

void *QUEUE::insert_first(void *newjob)
{
    Q_LINK *ql;

    ql = (Q_LINK *)free_q_links.allocate();
    ql->data = newjob;
    if (head==NULL)
        tail = ql;
    else
        head->prev = ql;
    ql->next = head;
    ql->prev = NULL;
    head = ql;
    return newjob;
}

void *QUEUE::insert_last(void *newjob)
{
    Q_LINK *ql;

    ql = (Q_LINK *)free_q_links.allocate();
    ql->data = newjob;
    if (head==NULL)
        head = ql;
    else
        tail->next = ql;
    ql->prev = tail;
    ql->next = NULL;
    tail = ql;
    return newjob;
}
}
```

Modeling Analogical Reasoning with DIS

```
void *QUEUE::remove(void *job)
{
    Q_LINK *cur=head;

    while (cur != NULL && cur->data != job)  cur = cur->next;
    if (cur == NULL) return job;

    if (cur==head)
        head = cur->next;
    else
        (cur->prev)->next = cur->next;

    if (cur==tail)
        tail = cur->prev;
    else
        (cur->next)->prev = cur->prev;

    free_q_links.free(cur);

    return job;
}
```

Appendix C. Memory management, FREELIST.H and FREELIST.CPP

FREELIST.H

```
/*          FREELIST.H

FREE_LIST class, manages allocation and freeing of elements
of the same class efficiently.  If an element is allocated,
it is never freed back to the system, but it may be returned
back to free_list to be available for the next allocation.
Elements are allocated in chunks.

Written by Oleg Ostrozhansky, December 1993
*/

#ifndef _FREELIST_DEFINED_
#define _FREELIST_DEFINED_

#define DEF_ALLOCAT    10      // default is to allocate in chunks of
                               // 10 elements

class FREE_LIST {
public:
    FREE_LIST(unsigned size, unsigned num = DEF_ALLOCAT);    // constructor
    /* arguments: size of each element, and optionally number of
       elements in each chunk */
    void *allocate();
    /* returns a pointer to a free element, or NULL if unsuccessful */
    void free(void *);
    /* adds an element to the free list */

private:
    void **head;      // head of the free list
    unsigned elem_size; // size of each element
    unsigned chunk;   // number of elements to allocate each time when
                       // out of free elements in the list
};

#endif
```

FREELIST.CPP

```

/*                                FREELIST.CPP

    Implementation of free_list class

    Written by Oleg Ostrozhansky, December 1993.

    If TRACK_MEMORY is defined, all allocated memory chunks are
    linked together into a linked list pointed to by mem_head
*/

#include <stdlib.h>
#include "freelist.h"

#ifdef TRACK_MEMORY
    void *mem_head = NULL;

    #define OFFSET sizeof(void *)
    #define RECORD_MEM(m) { *((void **)m - 1) = mem_head; \
                           mem_head = (void *)((void **)m - 1); }
#else
    #define RECORD_MEM(m)
    #define OFFSET 0
#endif

FREE_LIST::FREE_LIST(unsigned size, unsigned num)
{
    head = NULL;
    elem_size = size;
    chunk = num;
}

void *FREE_LIST::allocate()
{
    unsigned i;
    void **p;

    if (head == NULL) {
        /* allocate new chunk */
        head = (void *)((char *)malloc(elem_size*chunk+OFFSET)+OFFSET);
        RECORD_MEM(head);
        if (head == NULL) return NULL;
        p = head;
        for (i=0; i<chunk-1; i++) { // link new elements
            *p = (void *)((char *)p + elem_size);
            p = (void **) *p;
        }
        *p = NULL;
    }
    p = head;
    head = (void **) *p;
    return (void *)p;
}

void FREE_LIST::free(void *p)
{
    *((void ***)p) = head;
    head = (void **)p;
}

```

Appendix E. Simple example of DIS usage, EXAMPLE.CPP

```
/*                                EXAMPLE.CPP
    Demonstrates DIS usage
*/

#include <stdio.h>
#include "dis.h"

Object child_1("child_1"), child_2("child_2");
Object parent_1("parent_1"), parent_2("parent_2"), grand_parent("grand");

main()
{
    child_1.set_superclass(&parent_1);
    child_2.set_superclass(&parent_1);

    parent_1.set_attrib("color", "blue");
    parent_2.set_attrib("color", "green");

    printf("1: child_1 color: %s\n", child_1.get_attrib("color", NULL));
    child_1.set_attrib("color", "purple");
    printf("2: child_1 color: %s\n", child_1.get_attrib("color", NULL));
    printf("3: child_2 color: %s\n", child_2.get_attrib("color", NULL));
    child_2.set_superclass(&parent_2);
    printf("4: child_2 color: %s\n", child_2.get_attrib("color", NULL));
    printf("5: child_1 color: %s\n", child_1.get_attrib("color", NULL));
    child_1.rm_attrib("color");
    printf("6: child_1 color: %s\n", child_1.get_attrib("color", NULL));

    return 0;
}
```

Output:

```
1: child_1 color: blue
2: child_1 color: purple
3: child_2 color: blue
4: child_2 color: green
5: child_1 color: purple
6: child_1 color: blue
```

Appendix F. Prototype layer, PROTOT.H and PROTOT.CPP

PROTOT.H

```
/*          PROTOT.H

    PROTOTYPE layer of DIS

    Written by Oleg Ostrozhansky, October 1994
*/

#define F_OBJ_PTR      1      /* function requires ptr to the object */
#define F_GO_ON        2      /* lookup the result */

#define Printf      (void *)_Printf

typedef struct {
    void *func;
    char *arg;
    int flags;
} func_struct;

func_struct *func(void *f, void *arg, int flags);
void rm_func(func_struct *f);
char *invoke(Object *inst, char *func);
char *_Printf(char *);
```

PROTOT.CPP

```
/*          PROTOT.CPP

    PROTOTYPE layer of DIS, implementation

    Written by Oleg Ostrozhansky, October 1994
*/

#include <stdio.h>
#include "freelist.h"
#include "dis.h"
#include "protot.h"

static FREE_LIST free_func_structs(sizeof(func_struct), 10);

func_struct *func(void *f, void *arg, int flags)
{
    func_struct *x = (func_struct *)free_func_structs.allocate();

    x->func = f;
    x->arg = (char *)arg;
    x->flags = flags;
    return x;
}

void rm_func(func_struct *f)
{
    free_func_structs.free(f);
}
```

Modeling Analogical Reasoning with DIS

```
char *invoke(Object *inst, char *func)
{
    char name[NAMESIZE];
    char *script_ptr=func, *c;
    func_struct *f;
    char *return_val=NULL;

    while (*script_ptr != '\0') {
        // get name of the next function to invoke
        c = name;
        while (*script_ptr != ' ' && *script_ptr != '\0')
            *c++ = *script_ptr++;
        while (*script_ptr == ' ')
            script_ptr++;
        *c = '\0';
        // lookup the pointer to the function
        if ((f=(func_struct *)inst->get_attrib(name)) == NULL) {
            printf("undefined function: %s\n", name);
            continue;
        }
        // execute the function
        return_val = (f->func == NULL)? f->arg :
            (f->flags & F_OBJ_PTR)?
                (*((char *(*)(Object *, char *))(f->func))(inst, f->arg)) :
                (*((char *(*)(char *))(f->func))(f->arg);
        // if GO_ON, recursively process the result
        if (f->flags & F_GO_ON)
            return_val = invoke(inst, return_val);
    }
    return return_val;
}

char *_Printf(char *s)
{
    printf(s);
    return NULL;
}
```

Appendix G. Running into an obstacle, OBSTACLE.CPP

```

/*
    Demonstration program for DIS (Dynamic Inheritance System)

    Written by Oleg Ostrozhansky, August 1994
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freelist.h"
#include "dis.h"
#include "protot.h"

char *obstacle_do(Object *, char *);

main()
{
    Object *my_diff_problem;
    Object *obstacle, *c, *diff_problem;
    int i;

    puts("***  __DATE__  " "  __TIME__  " ***");

    c = new_class("Problem");
    c->set_attrib("do", func(NULL, "solve", F_GO_ON));
    c->set_attrib("solve", func(Printf, "solving a problem...", 0));
    c->set_attrib("done", func(Printf, "Done!", 0));

    diff_problem = new_class("Diff equation");
    diff_problem->set_attrib("solve", func(Printf, "differentiating...\n", 0));
    diff_problem->set_attrib("push_harder", func(Printf, "integrating...\n", 0));
    diff_problem->set_attrib("go_around", func(Printf, "reading book...\n", 0));
    diff_problem->set_superclass(c);

    obstacle = new_class("Obstacle");
    obstacle->set_attrib("do", func((void *)obstacle_do, NULL, F_OBJ_PTR));

    my_diff_problem = new_class("my problem");
    my_diff_problem->set_superclass(diff_problem);

    for (i=1; i<5; i++)
        invoke(my_diff_problem, "do");

    puts("Hmm.  No progress.  I think it is becoming an obstacle.");

    my_diff_problem->set_superclass(obstacle);
    while (!invoke(my_diff_problem, "do"))
        ;

    puts("Got it!");

    return 0;
}

char *obstacle_do(Object *inst, char *x)
{
    void *p = (void *)0;

```

Modeling Analogical Reasoning with DIS

```
x = x;                // to avoid compiler warning; arg is not used

p = inst->get_attrib("stuck_count", PRIVATE);

if ((int)p<5) {
    inst->set_attrib("stuck_count", (char *)p+1);
    invoke(inst, "push_harder");
    return (char *)0;
}

puts("Pushing harder didn't work.  Have to go around.");
inst->rm_attrib("stuck_count");
invoke(inst, "go_around");
return (char *)1;
}
```

Output:

```
*** Nov 27 1994  22:10:49 ***
differentiating...
differentiating...
differentiating...
differentiating...
Hmm.  No progress.  I think it is becoming an obstacle.
integrating...
integrating...
integrating...
integrating...
integrating...
Pushing harder didn't work.  Have to go around.
reading book...
Got it!
```

Appendix H. Adjusting a quantitative value, ADJUST.CPP

```

/*                                ADJUST.CPP

    Demonstration program for DIS (Dynamic Inheritance System),
    throwing a ball and making it travel exactly the specified distance
    by adjusting the effort

    Written by Oleg Ostrozhansky, November 1994
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freelist.h"
#include "dis.h"

#define throw throw_obj
    // throw is a reserved word in C++

// results of estimation
#define NO_RESULT      0
#define TOO_MUCH      1
#define NOT_ENOUGH    2
#define CLOSE         4
#define WAY_OFF       8

/* Real World Interface */
struct {
    int velocity;
    int coordinate;
} real_world_if;

Object *do_smth, *do_quant, *estimate, *throw, *throw_ball_now;

void build_class_structure(void);
void engine(void);
void real_world(void);
void brain(void);
int do_smth_do(Object *obj, void *x);
int do_quant_do(Object *obj, void *x);
int do_quant_more(Object *obj, void *x);
int do_quant_less(Object *obj, void *x);
int estimate_think(Object *obj, void *current_value, void *compare_to);
int throw_do_it(Object *obj, void *x);

main()
{
    puts("***  " __DATE__  "  " __TIME__  "  ***");

    build_class_structure();

    while (1) {
        engine();
        real_world();
        brain();
    }
}

```

Modeling Analogical Reasoning with DIS

```
}

void build_class_structure()
{
    do_smth = new_object("do_smth");
    do_smth->set_attrib("do", (void *)do_smth_do);

    do_quant = new_object("do_quant");
    do_quant->set_attrib("do", (void *)do_quant_do);
    do_quant->set_attrib("do_more", (void *)do_quant_more);
    do_quant->set_attrib("do_less", (void *)do_quant_less);
    do_quant->set_attrib("result", NO_RESULT);

    estimate = new_object("estimate");
    estimate->set_attrib("think", (void *)estimate_think);

    throw = new_object("throw");
    throw->set_attrib("do_it", (void *)throw_do_it);
    throw->set_attrib("effort", (void *)10);
    throw->set_attrib("delta_effort", (void *)2);

    throw_ball_now = new_object("throw_now");
    throw_ball_now->set_attrib("goal", (void *)50);

    do_quant->set_superclass(do_smth);
    throw_ball_now->set_superclass(do_smth);
    throw_ball_now->set_superclass(throw);
}

void engine()
{
    DIS_FUNC(throw_ball_now->get_attrib("do", NULL))(throw_ball_now, NULL);
}

void real_world()
{
    real_world_if.coordinate = 2*real_world_if.velocity;
}

void brain()
{
    int delta, result;

    delta = (int)throw_ball_now->get_attrib("delta_effort", NULL);
    throw_ball_now->set_superclass(estimate);
    // estimate relatively to delta
    result = DIS_FUNC2(throw_ball_now->get_attrib("think"))
        (throw_ball_now, (void *)real_world_if.coordinate, (void *)delta);
    throw_ball_now->set_attrib("result", (void *)result);
    throw_ball_now->rm_superclass(estimate);

    if (throw_ball_now->get_attrib("Quant", PRIVATE) == NULL) {
        // throw_ball_now is not a subclass of do_quantitative yet
        throw_ball_now->set_attrib("Quant", "effort"); // balance "effort"
        throw_ball_now->set_superclass(do_quant);
    }
}
```

Modeling Analogical Reasoning with DIS

```
int do_smth_do(Object *obj, void *x)
{
    x=x;
    return DIS_FUNC(obj->get_attrib("do_it", NULL))(obj, NULL);
}

int do_quant_do(Object *obj, void *x)
{
    int result, delta;
    char delta_name[NAMESIZE+7] = "delta_";
    x=x; // to avoid compiler warning

    result = (int)obj->get_attrib("result", NULL);
    if (result == NO_RESULT) {
        DIS_FUNC(obj->get_attrib("do_it", NULL))(obj, NULL);
        return 0;
    }
    strcpy(delta_name+6, (char *)obj->get_attrib("Quant", NULL));
    delta = (int)obj->get_attrib(delta_name, NULL);

    if (result & WAY_OFF)
        delta *= 2;
    else if (result & CLOSE)
        delta /= 2;
    if (delta == 0)
        delta = 1;

    obj->set_attrib(delta_name, (void *)delta);

    DIS_FUNC(obj->get_attrib((result & TOO_MUCH? "do_less" : "do_more"), NULL))
        (obj, NULL);

    return 0;
}

int do_quant_more(Object *obj, void *x)
{
    char delta_name[NAMESIZE+7] = "delta_";
    int delta;
    x=x; // to avoid compiler warning

    strcpy(delta_name+6, (char *)obj->get_attrib("Quant", NULL));
    printf("Doing more of %s\n", delta_name+6);
    delta = (int)obj->get_attrib(delta_name, NULL);

    obj->set_attrib(delta_name+6,
        (void *)((int)obj->get_attrib(delta_name+6, NULL) + delta));

    DIS_FUNC(obj->get_attrib("do_it", NULL))(obj, NULL);
    return 0;
}

int do_quant_less(Object *obj, void *x)
{
    char delta_name[NAMESIZE+7] = "delta_";
    int delta;
    x=x; // to avoid compiler warning

    strcpy(delta_name+6, (char *)obj->get_attrib("Quant", NULL));
    printf("Doing less of %s\n", delta_name+6);
    delta = (int)obj->get_attrib(delta_name, NULL);
```

Modeling Analogical Reasoning with DIS

```
obj->set_attrib(delta_name+6,
    (void *)((int)obj->get_attrib(delta_name+6, NULL) - delta));

DIS_FUNC(obj->get_attrib("do_it", NULL))(obj, NULL);
return 0;
}

int estimate_think(Object *obj, void *current_val_arg, void *compare_to_arg)
{
    #define current_value (int)current_val_arg
    #define compare_to (int)compare_to_arg
    int goal, result;

    goal = (int)obj->get_attrib("goal", NULL);

    if (current_value > goal + compare_to*3)
        result = TOO_MUCH | WAY_OFF;
    else if (current_value > goal + compare_to)
        result = TOO_MUCH;
    else if (current_value > goal)
        result = TOO_MUCH | CLOSE;
    else if (current_value == goal) {
        puts("Done!");
        exit(0);
    } else if (current_value >= goal - compare_to)
        result = NOT_ENOUGH | CLOSE;
    else if (current_value > goal - compare_to*3)
        result = NOT_ENOUGH;
    else
        result = NOT_ENOUGH | WAY_OFF;

    return result;

    #undef current_value
    #undef compare_to
}

int throw_do_it(Object *obj, void *x)
{
    x=x; // to avoid compiler warning

    real_world_if.velocity = (int)obj->get_attrib("effort", NULL);
    printf("Throwing with effort %d\n", real_world_if.velocity);
    return 0;
}
```

Output:

```
*** Nov 28 1994 19:50:51 ***
Throwing with effort 10
Doing more of effort
Throwing with effort 14
Doing more of effort
Throwing with effort 22
Doing more of effort
Throwing with effort 26
Doing less of effort
Throwing with effort 24
Doing more of effort
Throwing with effort 25
Done!
```